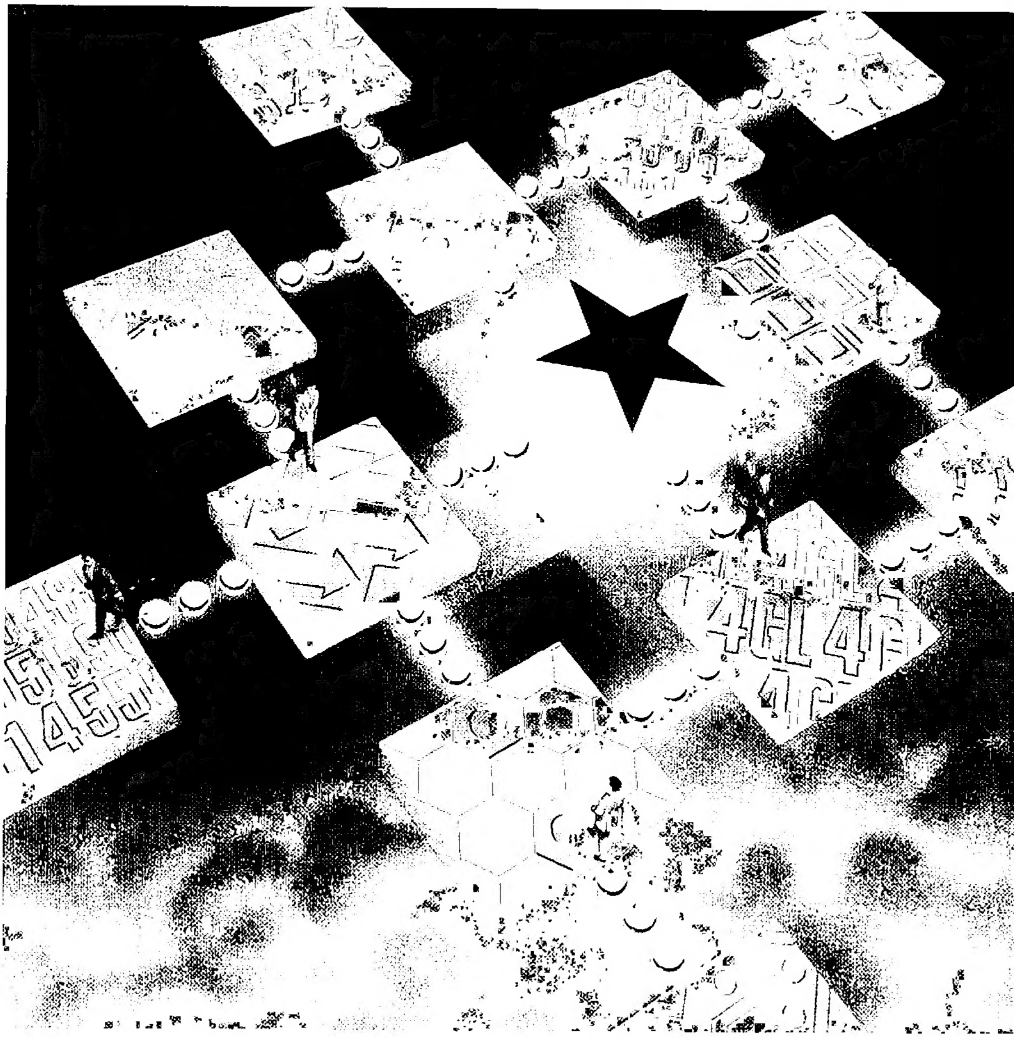
APL IS EASY!



"A straightforward introduction to the language ... will give you an excellen working knowledge and a very solid base upon which you can further develop your skills." — *Byte*

anugistics™
working as one

10 9 8 7 6 5 4 3

Contents

Appendices

Introduction

APL (A Programming Language) is a concise and powerful computer programming language. This book was written to help you learn the APL language. The book introduces the basic concepts of APL and teaches introductory techniques you can use to start programming in APL. Although you will benefit most from this book if you are familiar with computers and programming, you can learn the basics of APL programming even if you are a computer novice.

Kenneth E. Iverson originally invented APL as a mathematical notation in the late 1950s. In the mid-1960s, the notation was implemented as a programming language for use by IBM's central research staff at the Thomas J. Watson Research Laboratory. As APL developed, users discovered that APL offered significant productivity advantages when compared with more traditional programming languages such as FORTRAN and COBOL. APL is now used by companies worldwide on a wide variety of hardware platforms and operating systems.

APL Is Easy! has been adapted to fit most commercial APL systems, including STSC's APL★PLUS systems, IBM's APL2, and I. P. Sharp Associates' SHARP APL, and concentrates on elements of the language common to these systems. If, however, something does not produce a result you expected, or does not work as shown in this book, refer to your system documentation or an experienced user of your APL system.

assumptions

The book assumes that:

- You have installed APL★PLUS on your computer and that you have turned on your computer and started APL.
- You know how to use your keyboard.

If you have not installed APL★PLUS, read the installation instructions provided with the system to learn about installing and starting APL★PLUS on your computer.

If you use an APL system other than APL★PLUS, be sure to read the system documentation so that you are familiar with that system's keyboard.

organization

APL Is Easy! contains 10 chapters, appendices, a glossary, and an index. The chapters contain the following information:

- Chapter 1 explains how to do simple arithmetic and store data using APL.
- Chapter 2 explains how words and sentences are handled in APL, and also introduces other ways to do arithmetic.
- Chapter 3 explains ways to analyze data.
- Chapter 4 deals with ways to analyze and manipulate data.
- Chapter 5 introduces the concept of programs and data.
- Chapter 6 explains how to write and correct programs.
- Chapter 7 provides additional programming tools, including the use of friendly interactive programs.
- Chapter 8 explains how to format data with the format function and describes a system function that gives you more control over the way the computer prints information.
- Chapters 9 and 10 deal with the two storage facilities of APL.

Appendix A contains the answers to the exercises and explanations of some of the more subtle points. It is important that you work through the exercises, since that is the best way to learn APL. Appendix B contains common APL error messages, their causes, and solutions. Appendix C contains the programs used in this book.

Each chapter contains an introduction and a summary. Most chapters include exercises. The inside margin contains important keywords to help you locate a topic easily.

conventions

APL examples are shown in APL font (italic capital letters plus special APL symbols). Each example includes two parts: the information you type and the system response.

In APL, the information you type is indented six spaces. The system response is printed at the left margin; for example:

```
      9÷3           Your entry.
3                   The system response.
```

The system always returns to the six-space indent to wait for you to enter something new.

Throughout this book, the word enter means that you should do two things: type input and press the Return key or the Enter key.

saving your work

Before starting the exercises at the end of a chapter, you should save your workspace — the area that contains your work. If you are using a personal computer, be sure the correct floppy disk is in the disk drive. Enter:

```
      )SAVE MYWORK
```

You can use any name in place of *MYWORK*, as long as the name begins with a letter. The length of the name depends on the system you are using; refer to the system documentation to find out how many letters your names can contain.

After you save your workspace onto disk, enter *)CLEAR* to clear the area. Then try to solve the exercises in the chapter. When you finish with the exercises, you may want to save this additional work in a different workspace; for example:

```
      )SAVE EXERCISE
```

You can retrieve your work by entering a statement like:

```
        )LOAD MYWORK
```

or

```
        )LOAD EXERCISE
```

The name that you type after *)LOAD* depends on the name you gave the workspace when you entered the *)SAVE* command.

using the supplied programs

The programs that appear in the later chapters of this book, including those in the exercises, are in the *LESSONS* workspace supplied with the APL★PLUS System. Appendix C lists the APL programs for the *LESSONS* workspace. If your APL system does not have this workspace, enter the programs as they appear in Appendix C.

To copy the programs to your system, type

```
        )COPY  LESSONS pgmname
```

where *pgmname* is the name of the program you want to copy. For example, to copy the *AVG* program from the *LESSONS* workspace, enter:

```
        )COPY LESSONS AVG
```

If you are using the APL★PLUS System and you receive the message, *WS NOT FOUND*, try inserting one of the other disks supplied with the system into your primary disk drive. Then enter the command again.

correcting mistakes

As you work through the exercises, you will probably make typing errors and receive error messages (such as *LENGTH ERROR*). The following information explains how you use input line editing to make corrections.

- **Interrupting Input**
  If you type a line incorrectly and notice it *before* you press Return (or Enter), you can fix the mistake. The simplest thing to do is to interrupt your input, which causes the computer to ignore the entire line and return you to the six-space indent.

  To interrupt input, press the Ctrl-Break keys. (Other APL systems use slightly different techniques. If you do not know which keys to use, refer to the system documentation or ask an experienced user.)

- **Backspace**
  To delete characters at the end of a line, you can use another technique. Hold down the Backspace key until you erase all the characters you want. Then, type in the correct information, keeping the first part of the line.

- **Delete**
  Another way to delete characters is to use the Delete (Del) key. Use the cursor-movement keys (usually the keys marked with arrows) to move to the mistake, then press the Delete key. The system deletes the character *above* the cursor.

Take a few minutes to practice these techniques by typing long lines and making changes in them.

1
Arithmetic and Data Storage

In this chapter, you will learn:

- how to use APL to do arithmetic with single numbers, lists of numbers, and tables of numbers
- how to store these numbers in the computer's memory
- about the shape, reshape, and catenate functions.

At the end of this chapter, there are exercises for you to do. Try to solve at least some of the problems, so that you can gain experience with APL. Solutions and explanations are in Appendix A.

Simple Arithmetic

Unlike many other programming languages, you can be immediately productive with APL. In its simplest form, you can use APL like a desktop calculator. But there are a few differences between APL and a desktop calculator: how you handle negative numbers, the order in which APL executes arithmetic statements, and the ability to work with lists of numbers.

Using APL Like a Desk Calculator

immediate execution mode

In its usual state, you can use APL like a desk calculator. This state is called immediate execution mode, since the system immediately gives you an answer to what you type in. When you enter these arithmetic problems

```
      128+307
      832-299
      66÷2
      4×12
```

the system display looks like this:

```
      128+307
435
      832-299
533
      66÷2
33
      4×12
48
```

Notice that after each entry you make, the system displays its response — in these examples, the answer to a mathematical problem. Throughout this book, you will see many instances of entries and system responses. In every case, your entries appear with the six-space indent and the system responses appear at the left margin.

Negative Numbers

In regular arithmetic notation, you use the same symbol to represent the subtraction operation (2 - 4) and to indicate a negative number (-2).

In APL, you use two different symbols for these two functions. To perform subtraction, you use the familiar middle minus (hyphen) sign. For example, when you enter

```
      10-4
```

the system displays:

```
6
```

high minus

To indicate a negative number, use the high minus sign. The high minus is usually located on the 2 key. You type it by pressing the Shift or Alt key and then pressing the 2 key. For example, using the high minus sign, enter:

```
      ¯3+8
```

The system responds with:

```
5
```

If you put the hyphen to the left of a number, it changes the number into a negative number. For example, when you enter:

```
      -4
```

The system displays:

```
¯4
```

Or, if you put the hyphen to the left of a negative number, it changes to a positive number:

```
      -¯3
3
```

But now try entering:

```
      ¯-4
```

The system responds with:

```
SYNTAX ERROR
      ¯-4
       ^
```

The system displays this error message because you cannot separate a high minus sign from the number it belongs to — you have made an error in forming the statement. The high minus must appear next to the number.

Order of Execution

You can enter two or more arithmetic functions in the same line. For example, enter:

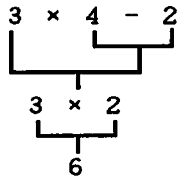
```
      3×4-2
```

The system displays:

```
6
```

right-to-left execution

You might expect the answer to be 10, because in regular arithmetic, you perform multiplication before addition and subtraction. However, APL has so many different functions that it would be confusing to have a set of rules about which functions to perform first. The system uses one easy rule when performing arithmetic — it always analyzes a statement right to left. That is, it evaluates the function at the far right of the line *first*, then moves left to the next function. The system evaluated the example above as follows:



Try a few more examples. Enter:

```
      6÷2+1
      8+3×2
```

The system display should look like:

```
      6÷2+1
2

      8+3×2
14
```

The system also works this way for statements with a large number of functions in them. For example, to find the sales tax on the purchase of three items priced at $1.32, $4.10, and $.78 (assuming a tax rate of 5%), you enter:

```
      .05 × 1.32 + 4.10 + .78
```

The system displays

```
0.31
```

since $1.32 + $4.10 + $.78 equals $6.20, and $6.20 × .05 equals $.31.

using parentheses

To change the order of execution (the way the system evaluates the statement), use parentheses to surround the expression you want the system to execute first. The system treats the contents of the parentheses as a single value, and therefore must calculate this value before using it. The following examples show how using parentheses in expressions affect the results.

```
      (6÷2)+1
4
```

```
      6÷2+1
2

      (8+3)×2
22

      8+3×2
14
```

Now, imagine that you need to buy new carpet for three rooms in your home. The rooms are 12 by 15, 20 by 15, and 15 by 12. To find the total number of square feet in the three rooms, you multiply the length and the width of each room and then add those results. In APL, you enter

```
      (12×15)+(20×15)+(15×12)
```

and the system displays

```
660
```

since (180) + (300) + (180) equals 660.

Handling Lists of Numbers

vectors and scalars

One of the unique features of APL is that a single function can work on long lists of numbers. A list of numbers, like 7 5 3, is called a vector. A single number is called a scalar. Try entering a list of numbers to see what happens. When you enter a list of numbers, leave at least one space between each of the numbers.

When you type

```
      7 5 3+7 4 2
```

the system displays:

```
14 9 5
```

What happened? APL★PLUS added the first number on the left of the plus sign to the first number on the right, the second number on the left to the second number on the right, and so on.

Try the following examples. Remember, you enter the information that is shown on the lines that begin with a six-space indent. The system responds by displaying the information that is shown on the lines that begin at the left margin.

```
      7 5 3×7 4 2
49 20 6

      7 5 3÷7 4 2
1 1.25 1.5

      8 6-6 4
2 2
```

Now enter:

```
      4 5+1 2 3
```

The system responds with:

```
LENGTH ERROR
      4 5+1 2 3
          ^
```

Since APL★PLUS cannot process the statement, it tells you why. It tells you that it cannot perform the addition because the length of the vector to the right of the plus (+) (emphasized with the ^) is not equal to the length of the vector to the left of the plus (+). Two vectors must have the same *length* (that is, contain the same number of elements) for APL to combine them.

operations with vectors and scalars

You can also perform operations that mix vectors with scalars (single numbers). Try:

```
      8 10 12+3
```

The system responds with:

```
11 13 15
```

The system added the single number (scalar) 3 to each individual number in the vector. If a scalar appears on one side of a function symbol (like + or ÷) and a vector appears on the other side, the scalar is added to (or subtracted from, multiplied by, and so on) each number in the vector. This behavior is known as scalar extension in APL. Try using scalar extension with the other arithmetic functions you have learned and you will see that the results are similar.

Storing Data in Variables

By now, you know that doing arithmetic in APL is similar to doing it on a desk calculator. Most calculators allow you to store data in one or more memory locations. APL stores data in variables and you can display the data in the variables.

In APL★PLUS , you store data in a variable using the assignment arrow (←), which is usually located on the left bracket ([ ) key. Try entering the following statements to create three variables *NUMBER*, *SALES*, and *GRADES*:

```
      NUMBER←1
      SALES←15.80 17.40 100.20
      GRADES←78 97 81 85 92
```

To display the data stored in a variable, enter the variable name. Variable names can consist of letters and numbers; however, the name begins with a letter. Try displaying the contents of the variables you just created. Enter *NUMBER*, then enter *SALES*, and then enter *GRADES*. The system display looks like this:

```
      NUMBER
1

      SALES
15.8 17.4 100.2

      GRADES
78 97 81 85 92
```

using variables with functions

You can use variables just as you use the numbers they contain. For example, to calculate the tax on the three items in the variable *SALES* (using a 5% tax rate), enter:

```
      SALES×.05
```

The system displays:

```
0.79 0.87 5.01
```

Or, to double the sales, enter:

```
      2×SALES
```

The system displays:

```
31.6 34.8 200.4
```

You can also add variables to other variables (or subtract, divide, and so on). For example, to add the variables *NUMBER* and *SALES*, you enter:

```
      NUMBER + SALES
```

The system displays:

```
16.8 18.4 101.2
```

Variables retain their values until you assign new data to them, as the following examples show.

```
      NUMBER
1
      NUMBER←2
      NUMBER + NUMBER
4
      NUMBER
2
      NUMBER←1
      NUMBER
1
```

You can use a variable and reassign it in the same line; for example:

```
      NUMBER←NUMBER+4
      NUMBER
5
```

Determining the Size of a Variable

APL provides a useful function — shape (ρ) — to help you determine how many numbers are in a variable. The shape of a variable is the number of elements in it. The shape function (ρ) is especially helpful if you work with variables that contain many numbers. For example, to find the shape of the variable *GRADES*, enter:

```
      ρGRADES
```

The system displays:

```
5
```

If you display the *GRADES* variable, you see that it contains five elements:

```
      GRADES
78 97 81 85 92
```

Create the following variable

```
      SALES1←200 175 125.50 421.75 89 23
304 89.95 256.72 78 21 78.23 75.21
```

and then display its shape:

```
      ρSALES1
13
```

Note: If you fill a line on the screen, APL★PLUS automatically moves the cursor to a new line. Do not press the Enter key to move the cursor to the next line.

Combining Variables

Another useful APL function is catenate. This function joins variables to make one large variable. The comma (,) is the symbol for catenation. Try using the catenate function (,). Enter the information shown on the lines that begin with a six-space indent. The system responds by displaying the information shown on the lines that begin at the left margin.

```
      SALES2←SALES,SALES1
      ρSALES
3
      ρSALES1
13
      ρSALES2
16
      SALES2
15.8 17.4 100.2 200 175 125.5 421.75 89 23
304 89.95 256.72 78 21 78.23 75.21

      ONE←1
      TWO←ONE,ONE
      TWO
1 1
      ρTWO
2
```

You can see that the shape function (ρ) is very useful for counting the elements in a long list of numbers produced using the catenate function (,).

Arithmetic with Tables of Numbers

You can use both scalars (single numbers) and vectors (groups of numbers) with arithmetic functions, and you can assign scalars and vectors to variables.

Another useful form for data is a table, such as the sales from different stores in different months. Table 1-1 is an example of data in table format.

Table 1-1. Data in Table Format

	Jan	Feb	Mar
Elm St. Store	10000	11000	10525
Lee Ave. Store	9000	9250	10000
Pine Ln. Store	6000	6100	6125
Ash St. Store	8250	10500	7500

Creating Tables

A matrix is a table of data, like the numbers in Table 1-1. A matrix has two dimensions: rows (going across) and columns (going down). You can make a table in APL using the shape function (ρ) in a new way. Enter:

```
      STORESALES←4 3ρ10000 11000 10525 9000
9250 10000 6000 6100 6125 8250 10500 7500
```

Now display the variable *STORESALES*:

```
      STORESALES
```

The system displays this data in table format:

```
10000 11000 10525
 9000  9250 10000
 6000  6100  6125
 8250 10500  7500
```

The numbers to the left of shape (ρ) tell the system how many rows and columns you want in the table. In the example above, the 4 and 3 to the left of shape (ρ) tells the system that you want to create a table (matrix) with four rows and three columns. You call shape (ρ) reshape when you use it with numbers on *both* the left and right sides. It reshapes the data on the right into the shape given on the left.

You can use the reshape function (ρ) to create vectors. For example, when you enter

```
        Q←4ρ7
```

the system creates a vector of four numbers (elements) — all 7s. To display the variable *Q*, enter:

```
        Q
```

The system responds with:

```
7  7  7  7
```

You can also use the reshape function (ρ) to create matrices with only *one* column. For example, when you enter

```
        R←3 1ρ5 10 15
```

the system creates a matrix with three rows and only one column. Display the variable *R*. Enter:

```
        R
```

The system displays:

```
 5
10
15
```

Or you can use the reshape function (ρ) to create a matrix with only one row. For example, when you enter

```
      Z←1 4ρ7
```

the system creates a matrix with one row and four columns. To display the variable Z, enter:

```
      Z
```

The system displays:

```
7 7 7 7
```

The variable Z *looks* like a vector, but it has a different shape. Use the shape function (ρ with data on the right side only) to display the shape of the variables Z and Q. The system display looks like this:

```
      ρZ
1 4
      ρQ
4
```

As you can see in the preceding example, the shape function (ρ) works with matrices as well as vectors. When you entered

```
      ρZ
```

the system responded with two numbers. These two numbers tell you that the data stored in the variable Z is a matrix that has one row and four columns. When you entered

```
      ρQ
```

the system responded with a single number. This number tells you that the data stored in the variable Q is a vector that has four elements.

Performing Arithmetic with Tables

This section explains how to use APL to do arithmetic with the data in matrices. Table 1-2 shows a matrix that contains the salaries of salesmen at three stores.

Table 1-2. Salesmen's Salaries

	Emp 1	Emp 2	Emp 3
Store 1	18000	17500	16500
Store 2	19000	21000	20000
Store 3	20000	17500	19500

To represent the salaries in APL, you use the reshape function (ρ) to create a matrix with three rows and three columns. Try it. Enter:

```
      SAL←3 3ρ18000 17500 16500 19000 21000
20000 20000 17500 19500
```

matrices
and scalars

Because your salesforce significantly exceeded their revenue targets for the past year, you decide to award them a 15% bonus. To calculate the 15% bonus for your staff, enter:

```
      BONUS←SAL×.15
```

Now display the *BONUS* variable.

```
      BONUS
 2700 2625 2475
 2850 3150 3000
 3000 2655 2925
```

APL★PLUS multiplies each number in the *SAL* matrix by .15 and stores the results in *BONUS*. Notice that the *BONUS* matrix has the same shape as the original matrix — three rows and three columns.

As a result of a fund-raising drive, each member of your staff has agreed to donate $500 to a local charity. To subtract $500 from each salesman's salary, you enter:

```
      SAL-500
```

The system displays:

```
 17500 17000 16000
 18500 20500 19500
 19500 17000 19000
```

Again, APL★PLUS uses the single number with each element of the matrix.

As you can see in these examples, arithmetic functions can use matrices (like *SAL*) and scalars (like 500) together.

matrices
and vectors

You cannot combine matrices and vectors, because the rank (or number of dimensions) is not the same. The problem is that APL★PLUS does not know how you want to add the numbers in the vector to the numbers in the matrix.

For example, in the statement

```
      SAL+8 9 7
```

do you want the system to add 8 to all the numbers in the first *row* or to all the numbers in the first *column*? Try entering:

```
      SAL+8 9 7
```

The system responds with:

```
RANK ERROR
      SAL+8 9 7
         ^
```

Since APL★PLUS cannot process the statement, it tells you why. The system tells you that it cannot perform the addition (emphasized with the ^) because the ranks differ.

matrices
and matrices

You can use matrices with other matrices, if they are the same shape. For example, the variables *SAL* and *BONUS* are both matrices with three rows and three columns. To calculate salaries plus bonuses for the salesforce for the entire year, you enter:

```
      SAL+BONUS
```

The system displays:

```
20700 20125 18975
21850 24150 23000
23000 20125 22425
```

Other Function Symbols with Double Uses

You probably wonder how rho (ρ) can perform two different tasks: it not only measures the size of a variable, but it also creates a vector or matrix.

arguments

In APL terminology, data to the left and right of a function are called arguments. You can use rho (ρ) with one argument (in particular, a *right* argument) to find the shape of a variable, or with two arguments (left and right arguments) to create a vector or matrix. You can use many APL functions this way.

APL functions with one argument are called monadic functions; functions with two arguments are called dyadic functions. Each function produces different results.

monadic and dyadic functions

Table 1-3 summarizes the rules for combining scalars, vectors, and matrices using the arithmetic functions you have learned so far. In the table, *fn* stands for any of the arithmetic functions (+, −, ÷, ×) and the arrow (←) identifies the result of the operation. For example, "scalar ← scalar *fn* scalar" means that when you use any of these functions with two scalars, the result is a scalar.

Table 1-3. Rules for Combining Scalars, Vectors, and Matrices

Result	Left Argument		Right Argument	
scalar	←	scalar	fn	scalar
vector	←	scalar	fn	vector
vector	←	vector	fn	scalar
vector	←	vector	fn	vector
matrix	←	scalar	fn	matrix
matrix	←	matrix	fn	scalar
matrix	←	matrix	fn	matrix

Many of the functions shown in the following chapters also follow these rules.

Note: Before you begin the exercises or go to the next section, be sure to save (store) the variables you have created so you can use them later. To save your variables, enter:

```
        )SAVE MYWORK
```

If you have difficulty saving your variables or you need more information, please refer to the Introduction.

Summary

In this chapter, you learned that:

- In APL, arithmetic calculations are performed much like you would do them by hand or with a calculator.

- In APL, a consistent order is used to execute functions on a line: always right to left. To change the order, use parentheses to enclose those operations you want APL to perform first.

- In APL, special symbols are used to perform other operations. You learned about three symbols: the assignment arrow (←), rho (⍴), and the comma (,).

  The assignment arrow (←) stores numbers by assigning them to a name. Scalars, vectors, and matrices can all be assigned to variables.

  Rho (⍴) can be used two ways: to measure the size of something (shape) or to create a vector or matrix (reshape).

  The comma (,) is used to join (catenate) variables.

- Many APL symbols serve two purposes. You can use them with a right argument only (that is, data on the right side of the function symbol) or with both left and right arguments (data on both sides of the symbol).

Exercises

A. Try some practice exercises to solve the problems before you do them on the computer. Some of the exercises produce error messages rather than actual answers. Explanations of these messages are in Appendix A, along with the solutions.

1. `3+1 2 3`
2. `6÷2`
3. `2-4`
4. `3×2 4`
5. `3 3×2 4`
6. `4+`
7. `3¯2`
8. `7 4 5-2`
9. `10÷0`
10. `5×2+6`
11. `4 6+3 5 7`
12. `(5×2)+6`
13. `5(2+6)`
14. `4÷1 2`
15. `-¯8`
16. `4÷2×1+1`

B. Create three variables as shown below:

```
A←2
B←1 4 7
C←2 3ρ0 1 2 3 4 5
```

Use these variables to do the exercises in this section.

1. `A+A`
2. `A+B`
3. `A+C`
4. `B+B`
5. `C+C`
6. `B+C`
7. `4+C`
8. `9 10 11-B`
9. `2 4-B`
10. `2×B`

C. Use the following variables for the problems in this section:

```
V1←0 0 1
V2←1 2 3 4
V3←2 1
```

1. `ρV1`
2. `ρV2`
3. `ρV1,V2`
4. `V1,5`
5. `ρV1,5`
6. `V2+V1,5`
7. `3ρ4`
8. `3 3ρV3`
9. `V3ρV1`
10. `5,V1`

D. You are at the grocery store and you buy the following items:

- 12 potatoes at 20 cents each
- 6 pears at 30 cents each
- 2 avocados at 50 cents each

Write an APL expression using two vectors to calculate the total cost of each item.

E. Write an APL expression to find the yearly interest on four savings accounts, given annual interest rates of 5%, 5.25%, 5.5%, and 6%. The four accounts contain $600, $700, $1000, and $950, respectively. (Hint: .05 is the same as 5%.)

F. To find the grades for students who scored 48, 39, 41, and 35 out of a possible 55 points, you use the following expression:

```
100×48 39 41 35÷55
```

The division part of the expression turns the grades into a ratio, and multiplying the ratios by 100 turns them into percentages.

Write an APL expression to find the percentage grades for students who scored 142, 167, 161, and 128 points out of a possible 175 points.

G. Write an APL expression to find the tax due on a taxable income of $7216. Assume the tax is $2116 plus 25% of the amount of taxable income exceeding $6000.

H. Your store sells hammers, wood, and nails. Last week, you sold 12 hammers, 9 pieces of wood and 3 nails; this week you sold 11, 12, and 6 items, respectively. Store this data in a matrix using one row for each week.

The prices for the three items were $3.25, $1.79, and $2.55 last week, but this week you raised them to $3.45, $1.85, and $2.75. Store these prices in a second matrix. Now, write an APL expression that calculates the total sales revenue by item for both weeks.

I. The vector *GRADES* contains students' grades from last semester (90, 82, 79, 92). Put the grades for this semester's students (93, 87, 80, 90, 78, 85) into the variable *GRADES* without losing last semester's grades. (Hint: Introduce a new variable *OLDGRADES*.)

J. You are the manager of a small store and you take inventory at the end of the month. You store the number of items in inventory in the variable *QUANT*:

```
      QUANT←210 51 34 27
```

You store the cost of a single unit of each item in the variable *COST*:

```
      COST←2.25 15.17 20.72 5.75
```

Write an APL expression that calculates the cost of inventory and assigns it to the variable *INVCOST*.

2

Character Data, More Arithmetic, and More APL Functions

In this chapter, you will learn:

- how APL handles and stores words and sentences
- how to use the maximum and minimum functions
- how to use the plus reduction and plus scan functions to add numbers and to produce a running total
- how to use the index function to pick out single elements of stored data
- how to use the catenate function with matrices
- how to use the count and roll functions to produce strings of numbers.

Character Data

In the previous chapter, you learned how to use numbers in APL. Numbers are known as numeric data. One of the special features of APL is that it can work with words and sentences as well as numbers. Letters, symbols, words, and sentences are known as character data in APL.

using single quotes

In APL, you distinguish character data from numeric data by enclosing characters in single quotes. On some keyboard layouts, the single quote mark is found on the K key. To enter a single quote, press and hold the Shift (or Alt) key and then press the K key. Try entering:

```
      'HELLO'
```

The system responds with:

```
HELLO
```

The system displays anything you put inside single quotes exactly as you enter it.

You can determine the shape of a group of characters using the shape function (ρ). Enter:

```
      ρ'HELLO'
```

The system displays:

```
5
```

APL also treats any blanks within the single quote marks as character data. Try the following examples. Enter the information that is shown on the lines that begin with a six-space indent. The system responds by displaying the information that is shown on the lines that begin at the left margin.

```
      ρ'HI THERE'
8
```

```
      ρ'I  SPY'
6
```

(Notice that there are *two* blanks between the words *I* and *SPY* in the second example.)

You can also use APL characters as character data. Try these examples. (On some keyboard latouts, Shift-H creates the ∆ symbol and Shift-1 creates the ¨ symbol.) Remember, you enter the lines with the six-space indent; the system responds with the lines at the margin.

```
      '***   ∆∆∆'
***   ∆∆∆
      '¨WHAT WAS THAT?¨'
¨WHAT WAS THAT?¨
```

open quote error

If you type only one quote, APL considers it an open quote error. APL systems react differently to this error; however, most systems stop until you correct it. To correct the error, you interrupt the system by pressing the Ctrl-Break keys and then re-entering the expression, or you can type a second quote and press the Enter key.

using quotes in character data

If you want to include a single quote as a character in a sentence, enter two single quotes one after the other. (Neither a double quote (") or a dieresis (¨) is the same as two single quotes.) Try the following examples (you enter the lines that begin with the six-space indent).

```
      'WHAT''S UP?'
WHAT'S UP?

      'DON''T SAY AIN''T.'
DON'T SAY AIN'T.
```

The system converts and displays the two single quotes as one quote. As a general rule, you must have an *even* number of quotes in a line to avoid the open quote error message.

using numbers as character data

You can also use digits as character data. For example, enter:

```
      '73'
73

      '1+1'
1+1
```

Notice that the system treats everything between the quotes as character data, including the plus sign.

You cannot, however, perform arithmetic with numbers you enter as character data. The system does not treat them as numbers. For example, when you enter

```
      3+'8'
```

the system displays:

```
DOMAIN ERROR
      3+'8'
        ^
```

In this example, the system displays a *DOMAIN ERROR* message because it is not in the domain of the plus function (+) to add numbers to characters. Entering `3+'8'` is like entering `3+'*'`— it is a meaningless statement.

Using Character Vectors

Groups of characters in a line, like the ones you have been entering, are called character vectors. You can assign character vectors to variables just like numeric vectors; for example:

```
      MESSAGE←'HELLO, WORLD.'
      MESSAGE
HELLO, WORLD.
```

using catenate with character data

You can also join character vectors to other character vectors with the catenate function (,). Try entering:

```
      PART1←'HELLO.  '
      PART2←'WHAT''S YOUR NAME?'
      PART1,PART2
```

The system displays:

```
HELLO.  WHAT'S YOUR NAME?
```

format function

In most APL systems, you cannot join character vectors directly to numbers. For example, when you enter

```
      'THE TOTAL IS ',27
```

the system displays:

```
DOMAIN ERROR
      'THE TOTAL IS ',27
                     ^
```

Just as it is not in the domain of the plus function (+) to add numbers and characters, it is not in the domain of the catenate function (,) to combine numbers and characters. You can join the data if you use the APL format function (⍕) to convert the number to character data. (This symbol is usually found on the single/double quote key; you generally type it as a combination with the Shift or Alt keys.) The format function (⍕) turns numbers into characters, which you can then catenate to other characters. Enter:

```
      'THE TOTAL IS ',⍕27
```

The system displays:

```
THE TOTAL IS 27
```

Using Character Matrices

You can use the reshape function (ρ) to change character vectors into character matrices; for example:

```
      TEXT←3 3ρ'HOWAREYOU'
      TEXT
HOW
ARE
YOU

      ANSWER←2 6ρ'FINE, THANKS'
      ANSWER
FINE,
THANKS
```

Notice that you must insert blanks in the proper places so that shorter lines have blanks at the end. Otherwise, the system fills the shorter lines with data you meant to appear on the next line.

In the next example, you create a character matrix using a string of three names: Jill, Jim, and Bob. The matrix is three characters high and four characters across, but you only enter 10 characters. The system uses the characters in groups of four (such as *JILL* and *JIMB*). When the system reaches the end of the character string, it returns to the beginning of the string to complete the matrix. In the example

```
      NAMES←3 4ρ'JILLJIMBOB'
      NAMES
JILL
JIMB
OBJI
```

the *OBJI* in Row 3 consists of the *OB* from the end and the *JI* from the beginning of the string (3 4ρ'*JILLJIMBOB*').

You can fix the matrix by inserting blanks after the names Jim and Bob:

```
      NAMES←3 4ρ'JILLJIM BOB '
      NAMES
JILL
JIM
BOB
```

Maximum

The maximum function (⌈) chooses the larger of two numbers. It compares the number on the right of the function with the number on the left and determines which number is larger. The symbol for the maximum function (⌈) is usually found on the S key. Try the following examples.

```
      8⌈10
10

      2.1⌈2
2.1
```

Like all APL functions, the maximum function (⌈) also works with data stored in variables. The system looks at the values in each variable and displays the larger value. Try these examples:

```
      ONE←1
      TWO←2
      ONE⌈TWO
2
```

You can store the result of the maximum function (⌈) in a variable. (This is true of *all* primitive APL functions.)

```
      B←7⌈19
      B
19
```

using maximum with vectors and matrices

You can also use the maximum function (⌈) with vectors and matrices. Try the following examples.

```
      1⌈3 1 0 6
3 1 1 6
      5 6 3 1⌈6 8 2 .5
6 8 3 1
```

```
      TABLE←2 3ρ1 2 3 4 5 6
      TABLE
1 2 3
4 5 6

      3⌈TABLE
3 3 3
4 5 6
```

Minimum

The minimum function (⌊) compares two numbers and displays the smaller of the two. The minimum symbol (⌊) is usually found on the D key.

The shapes of the minimum (⌊) and maximum (⌈) symbols make it easy to remember which function is which. Since the maximum symbol (⌈) extends at the top, relate it to the higher or larger number. Since the minimum symbol (⌊) extends at the bottom, relate it to the lower or smaller number.

Try using the minimum function.

```
      8⌊10
8

      1 3 4⌊4 3 2
1 3 2

      3⌊TABLE
 1 2 3
 3 3 3
```

Suppose you receive several orders for a discounted item, for which you have set a limit of 10 items per customer. Use the following APL statements to determine how many items can be delivered to each customer.

```
      DISCORDERS←12 15 7 9 17
      DELIVER←10⌊DISCORDERS
      DELIVER
10 10 7 9 10
```

Reduction

How would you write an APL statement that finds the largest number in a string of numbers? You could use the maximum function (⌈) and enter:

```
        10⌈12⌈2⌈72⌈16
```

maximum reduction

However, typing all of those numbers is time-consuming. APL provides a quick and easy way to do it. You combine the maximum function (⌈) with a slash (/), which has the same effect as inserting the symbols between each pair of numbers. This operator is called reduction. When you use the reduction operator with the maximum function, its full name is maximum reduction (⌈/). Try this example:

```
        ⌈/10  12  2  72  16
```

You can also use the maximum reduction function (⌈/) with variables; for example:

```
        ⌈/ORDERS
10.17
        ⌈/DISCORDERS
17
```

plus reduction

The reduction operator (/) works with other functions as well. When you use the reduction operator (/) with the plus function (+), it is the same as entering:

```
        2  +  3  +  4
```

Try it. Enter:

```
        +/2  3  4
9
```

times reduction

You can also use the reduction operator (/) with multiplication; this is called times reduction (×/):

```
          ×/2 3 4
24
```

reduction with matrices

You can also use the reduction operator (/) on a matrix. First, create a matrix called *NUMS*. Enter:

```
          NUMS ← 2 4 ρ 1 2 3 4 5 6 7 8
          NUMS
```

The system displays:

```
 1 2 3 4
 5 6 7 8
```

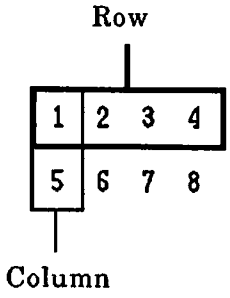
What is the shape of *NUMS*? Enter:

```
        ρNUMS
```

The system responds with

```
2 4
```

which tells you that *NUMS* is a matrix with two rows and four columns; that is:



When you use the plus reduction function (+/) on a matrix, the system adds the values in the *rows*. In the case of the *NUMS* matrix, the system adds the values 1, 2, 3, and 4 in the first row and displays the result — 10. Then the system adds the values 5, 6, 7, and 8 in the second row and displays the result — 26. Since

NUMS has two rows, the plus reduction function (+/) displays two numbers — one for each row of the matrix.

Try using the plus reduction function (+/) on this matrix. Type:

```
        +/NUMS
```

The system displays:

```
10  26
```

The system calculates the result as:

```
        1 + 2 + 3 + 4 = 10

        5 + 6 + 7 + 8 = 26
```

reduction across columns

But how do you use the plus reduction function (+/) to add the values in the *columns* of a matrix? You use ⌿ instead of /, or use [1] with the plus reduction function (+/). This produces a single value for each column in the matrix. Since *NUMS* has four columns, the plus reduction function (+⌿ or +/[1]) displays four numbers — one for each column of the matrix. Try it. Enter:

```
        +⌿NUMS
```

or

```
        +/[1]NUMS
```

The system displays:

```
   6   8 10  12
```

The system calculates the result as:

$$\begin{array}{rrrr} 1 & 2 & 3 & 4 \\ \underline{+\ 5} & \underline{+\ 6} & \underline{+\ 7} & \underline{+\ 8} \\ 6 & 8 & 10 & 12 \end{array}$$

When you want APL to add the values in the *rows*, you can use either

```
      +/NUMS
```

as you learned earlier, or you can use:

```
      +/[2]NUMS
```

Both statements mean the same thing in APL.

How would you find the grand total of the numbers in a table? The statement

```
      +/TABLE
```

gives you the result of adding the two rows. To find the grand total, you enter:

```
      +/6  15
21
```

To do this all in one statement, enter:

```
      +/+/TABLE
21
```

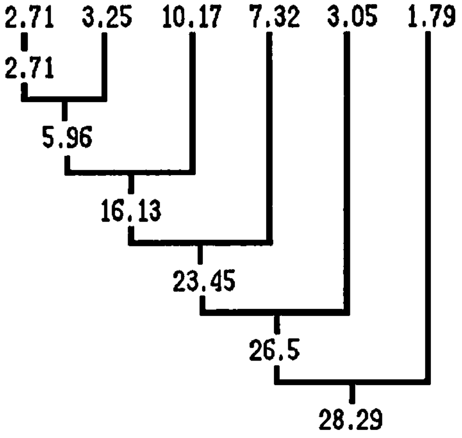
Scan

The plus reduction function (+/) lets you find the grand total of the numbers in a vector or a matrix. To find a *running* total, you use the scan operator (\) with the plus function (+). This function (+\) is called plus scan. Enter:

```
      ORDERS
2.71 3.25 10.17 7.32 3.05 1.79

      +\ORDERS
2.71 5.96 16.13 23.45 26.5 28.29
```

The result of this function shows the first number in the vector, then the sum of the first and second numbers, then the sum of the first, second, third, and so on. Visually, this looks like:



The last number in the result (28.29) is the grand total — the same result as +/*ORDERS.*

```
      +/ORDERS
28.29
```

Like the reduction operator, you can use the scan operator (\) with maximum, minimum, and other functions. Try using the times scan (×\) function. Enter:

```
      ×\2 3 4
2 6 24
```

Like the reduction operator, the scan operator (\) applies across the rows (\) or columns (⍀) of a matrix; for example:

```
      M←2 3 ⍴ 1 + ⍳6
      M
2 3 4
5 6 7

      ×\M
  2   6  24
  5  30 210

      ×⍀M
 2  3  4
10 18 28
```

Indexing

The index function has two symbols, the left and right brackets ([and]). You enter the position of the element you want (the index of the element) between the two brackets. The index ([*n*]) function is useful when you are working with variables that contain many elements.

Suppose you want to find the third element in the variable *DISCORDERS*. You can either display the contents of *DISCORDERS* and count to the third element or you can use the index ([*n*]) function. Enter:

```
      DISCORDERS
12 15 7 9 17

      DISCORDERS[3]
7
```

You can index with a vector of numbers:

```
      ORDERS
2.71 3.25 10.17 7.32 3.05 1.79

      ORDERS[1 3 5]
2.71 10.17 3.05
```

You can duplicate indices:

```
      ORDERS[2 2 2]
3.25 3.25 3.25
```

You can also use indexing to select the data in any order:

```
      VECT←9 10 11
      VECT
9 10 11
      VECT[3 2 1]
11 10 9
```

using index with variables

You can assign the result of indexing to a variable.

```
      BACKWARDS←VECT[3 2 1]
      BACKWARDS
11 10 9

      VECT
9 10 11

      NUMBER←VECT[2]
      NUMBER
10
```

replacing items in a vector

You can use indexing ([*n*]) with assignment (←) to replace individual items in a vector:

```
      VECT
9 10 11

      VECT[2]←3.14
      VECT
9 3.14 11

      VECT[3 1]←22 ¯44
      VECT
¯44 3.14 22
```

Using Index with Matrices

To find an element in a matrix, you must tell the system both the row *and* the column of the element you want. The first index is the row number; the second is the column number. Separate the two indices with a semicolon (;); for example;

```
      TABLE
 1 2 3
 4 5 6

      TABLE[2;1]
4
```

```
      TABLE[1;1]
1
```

You can also use more than one number when you index into a matrix. Enter:

```
      TABLE[1 2;1]
1 4
```

The system displays the numbers in Column 1 of Rows 1 and 2 . To display the first two rows *and* the first two columns, enter:

```
      TABLE[1 2;1 2]
 1 2
 4 5
```

If you leave out one of the indices, the system selects all of the numbers in the row or column you specify.

```
      TABLE[1;]
1 2 3

      TABLE[;3]
3 6
```

You can assign the results of these operations to variables.

```
      LITTLETABLE←TABLE[1 2;1 2]
      LITTLETABLE
 1 2
 4 5
```

You can also change individual elements of a matrix using indexing and assignment.

```
      TABLE[1;1]←25
      TABLE
 25  2  3
  4  5  6

      TABLE[2;2 3]←¯2 1.5
      TABLE
 25   2    3
  4  ¯2    1.5
```

The index you enter must correspond to an existing element in the vector or matrix. For example, the variable *ORDERS* is a vector that contains six elements. If you ask the system to display the seventh element of *ORDERS*, it displays a message that indicates the error — the index, in this case.

```
      ORDERS
2.71 3.25 10.17 7.32 3.05 1.79
      ORDERS[7]
INDEX ERROR
      ORDERS[7]
             ^

      ρORDERS
6
```

Using Index with Character Data

You can use indexing with character data, just as you would with numeric data. Try entering the following examples.

```
      MSG←'HI THERE'
      MSG[1 2]
HI

      MSG[5 6 7]
HER

      NAMES←3 4ρ'BILLJEANDAN '
      NAMES
BILL
JEAN
DAN

      NAMES[1 2;1]
BJ

      NAMES[3;1 2 1 2]
DADA
```

Catenation with Matrices

One function that is useful with character data is catenation. You learned how to catenate vectors earlier in this chapter; for example:

```
      V1←'HAM'
      V2←'BURGER'
      V1,V2
HAMBURGER
```

Because matrices have two dimensions, you must tell APL how to connect them: along the first dimension (rows) or along the last dimension (columns). If you do not specify a dimension, catenate selects the last dimension (columns).

Try to create and catenate the following matrices along the first dimension (rows). Enter:

```
      M1←3 3ρ'HISHERMY '
      M2←3 3ρ'CARBARTAR'
      M1,[1]M2
```

The `[1]`, which specifies the first dimension, indicates which element of the shape you want to change. The system displays:

```
HIS
HER
MY
CAR
BAR
TAR

      ρM1,[1]M2
6 3
```

Now, catenate *M*1 and *M*2 along the last dimension; that is, the columns. You do not need to specify the dimension explicitly, because catenate selects the last dimension by default. Enter:

```
      M1,M2
```

The system displays:

```
HISCAR
HERBAR
MY TAR
```

Count

You may want to work with a group of numbers without having to type them. The APL count function generates a string of consecutive whole numbers. Its symbol is an iota (ι). Try entering:

```
      ι7
1 2 3 4 5 6 7

      ι12
1 2 3 4 5 6 7 8 9 10 11 12
```

This function starts at 1 and counts, in ascending order, to the number you specify. You cannot use the count function (ι) with negative numbers or decimal numbers.

```
      ι¯4
DOMAIN ERROR
      ι¯4
      ^

      ι9.7
DOMAIN ERROR
      ι9.7
      ^
```

You can create almost any sequence of numbers with the count function (ι).

```
      .5+ι4
1.5 2.5 3.5 4.5

      2×ι3
2 4 6

      ¯5+ι6
¯4 ¯3 ¯2 ¯1 0 1
```

You can combine the count function (⍳) with the index function ([*n*]), as shown in the following example. Suppose you want to display the first four elements of *ORDERS*. You enter:

```
      ORDERS[1 2 3 4]
2.71 3.25 10.17 7.32
```

But you can also enter:

```
      ORDERS[⍳4]
2.71 3.25 10.17 7.32
```

Roll

Sometimes you may want to generate a list of *random* numbers rather than consecutive numbers. The roll function (?) generates a random number between 1 and the number you specify.

```
      ?10
4
      ?10
7
      ?6 6
1 5
```

(Since the roll function (?) generates random numbers, your system probably displays different answers than those shown above.) You can combine the roll function (?) with other number generators:

```
      ?⍳5
1 2 1 1 3
      ?3 2⍴6
 1 3
 5 5
 4 1
```

In the second example, the reshape function (⍴) creates a matrix of 6s (3 2⍴6), then the roll function (?) generates six random numbers, each between 1 and 6.

Note: Before you begin the exercises or go to the next section, be sure to save the variables you have created. To save your variables, enter:

```
      )SAVE MYWORK
```

If you have difficulty saving your variables or you need more information, please refer to the Introduction.

Summary

In this chapter, you learned that:

- In APL, you distinguish character data from numeric data by enclosing character data in single quotes. You can use any character (letters, numbers, or special symbols) in character vectors or character matrices.
- The format function (⍕) converts numbers to character data.
- You can join character vectors using the catenate function (,).
- Maximum and minimum are functions that compare two numbers. The maximum function (⌈) chooses the larger number; the minimum function (⌊) chooses the smaller number.
- You can combine maximum, minimum, and arithmetic functions with the slash (/) operator to form new functions. For example, you can combine the maximum function (⌈) with the reduction operator (/) to create the maximum reduction function (⌈/). These functions have the same effect as placing a function symbol between each element of a vector or matrix.
- You can also combine maximum, minimum, and arithmetic functions with the backslash (\) to form new functions called scans. If you combine the scan operator (\) with the plus function (+), the resulting plus scan function (+\) gives a running total of the numbers to its right.
- You can select elements from a vector or matrix using indexing. The index function ([*n*]) uses the left and right bracket ([and]) symbols. For a matrix, you must specify both the row and column number. Separate the row and column numbers with a semicolon (;).

- You can use , to catenate matrices as well as vectors. If you want the system to append rows rather than columns, include the dimension in the APL expression; for example, `MA,[1]M2`.

- You can produce a string of consecutive whole numbers using the count function (⍳). You can produce random numbers with the roll function (?).

Exercises

A. 1. Assign the characters ←→↑↓ to a variable named *ARROWS*. Display the variable.

2. Assign all the special APL characters to a variable called *CHARS* and display it.

3. Create a 3-by-5 character matrix called *NUMBERS*, where the first row contains the word *ONE*, the second *TWO*, and the third *THREE*. Remember, you form a character matrix as follows:

```
      ALPH←2 2ρ'ABCD'
      ALPH
AB
CD
```

Use spaces where necessary to make each row of the matrix five characters long.

4. Create a character matrix *ALPH* that looks like:

```
      ALPH
A
B
C
D
E
```

5. Create a character vector by entering:

```
      CV←'    '
```

What do you expect its shape to be? Enter ρ*CV* to check your answer.

6. Enter the following:

```
      A←'CAN''T BE'
```

What do you expect its shape to be? Enter ⍴*A* to check your answer.

7. Write an expression to find the fifth, second, and third elements of the vector *A* you created in Problem 6. All three letters should print together on the same line.

8. Define the following variables:

```
        B←'BLUE'
        C←'BELL'
        D←'JINGLE'
        E←'RINGER'
```

   What do you expect to see if you enter each of the following?

```
        B,C
        D,C
        C,E
```

   Now, enter the above expressions and compare the results to your expectations.

9. Enter the following:

```
        ALPH←'ABCDEFGHIJKLMNOPQRSTU
VWXYZ'
        SYMB←'∆∇|⋆○'
```

   a. Write an expression to return the 20th letter of the alphabet.

   b. Write an expression to return the fourth character in *SYMB*.

   c. Join *ALPH* and *SYMB* together.

   d. Display the shape of these two variables after you join them.

   e. Write an expression to print the first 12 letters of the alphabet.

f. Write an expression that returns three letters of the alphabet at random. (Hint: Use the roll function.)

10. Enter the expression:

```
10⍴'*'
```

What do you expect the system to display?

11. Enter the expression:

```
M←'THE NUMBER IS '
```

Now, join *M* with the number 20 on the same line.

12. Enter the expressions:

```
P1←'YOU HAVE SPENT '
P2←' DOLLARS.'
```

Write an expression that produces the following display:

```
YOU HAVE SPENT 10 DOLLARS.
```

(Hint: Use parentheses around ⍕10.)

B. Create the following variables and use them with the problems in Exercises B and C.

```
A←3 1 7 6
B←4 1 2 8
C←2 3⍴4 2 3 9 7 1
D←2 3⍴1 3 5 6 8 1
```

Write down the answers you expect; then check your answers by entering the expressions.

1. 4⌈*A*
2. 4⌊*A*
3. 2⌈*B*
4. 2⌊*B*
5. *A*⌈*B*
6. *B*⌈*A*
7. *A*⌊*B*
8. 3⌈*C*
9. 3⌈*D*
10. 4⌊*C*
11. 4⌊*D*
12. *C*⌈*C*
13. *C*⌈*D*
14. *C*⌊*D*

C. Use variables *A*, *B*, *C*, and *D* from Exercise B for the following problems. Again, write down the answers you expect; then check your answers by entering the expressions.

```
 1. +/A
 2. +/B
 3. +/C
 4. +/D
 5. ⌈/B
 6. ⌊/A
 7. ⌈/C
 8. ⌊/D
 9. +/[1]C
10. +/[1]D
11. +/+/C
12. +/+/D
13. ⌈/⌈/C
14. ⌊/⌊/D
```

D. Assign the following vector of bank transactions to the variable *TRANSACT*:

```
          TRANSACT←153.17 ¯20.00 17.00 ¯35.75
27.55 12.25
```

This is a record of your bank transactions for a month. Positive numbers are deposits; negative numbers are checks or withdrawals. Use this variable in the following exercises.

1. Write an expression to find the third transaction you made this month.

2. Write an expression to find the ending balance for your checking account for this month. (Assume that the beginning balance for the month was $0.00.)

3. Write an expression that shows the running balance of your account during the month.

4. Write an expression to subtract the last transaction from the first.

5. Your account earns 5% interest, computed monthly, based on the ending balance. Write an expression that shows how much interest you earned this month.

3

Selecting and Analyzing Data

In this chapter, you will learn to:

- use to select data from variables that meet specific conditions
- use the plus reduction function (+ /) to determine the *number* of items that meet a specific condition
- use the and reduction (∧ /) and or reduction (∨ /) function to determine if any or all of the data in a variable meet specific conditions
- use the and (∧) and or (∨) functions to select data from a variable that meets either one or two conditions.

Selecting Data from Lists

If you know the position of an element in a vector or matrix, you can use the index function ([*n*]) to display it. Sometimes, however, you may want to select elements without knowing their positions.

Suppose you want to know which deposits you made to your bank account during the past year that were greater than $80. The variable *DEP* lists each of your bank deposits during the period ($85, $125, $60, $45, and $89). Enter:

```
        DEP←85 125 60 45 89
```

To find out which deposits were greater than (>) $80, enter:

```
        DEP>80
```

(The greater than symbol (>) is usually located on the 7 key.) The system compares each value in *DEP* to 80. If the value is greater than 80, the system displays a 1; otherwise, the system displays a 0:

```
1 1 0 0 1
```

true or false results

The 1s and 0s mean true and false, or yes and no. If you compare the contents of the variable *DEP* with the results of the expression, you can see that the 1s correspond to the values you are looking for — the deposits greater than $80:

Deposit	$85	$125	$60	$45	$89
> $80 (1 = True)	1	1	0	0	1

Now you know that the first, second, and fifth elements of *DEP* are greater than 80. But what are the values? You use the compression function (/), discussed in the next two sections, to display the values.

Using Compression with Numeric Data

To select the numbers you want from *DEP*, the slash (/) uses the 0s (false) and 1s (true) that result from the expression *DEP*>80. This function is called compression because it keeps the numbers in the *DEP* variable that correspond to 1s (true), and leaves out the numbers that correspond to 0s (false).

In the following examples, combine the 0s and 1s that result from the expression *DEP*>80 with the compression function(/), Remember, you enter the lines with the six-space indent; the system responds with the lines at the margin.

```
      DEP>80
1 1 0 0 1

      1 1 0 0 1/DEP
85 125 89
```

There are three deposits greater than $80: $85, $125, and $89. Now try a simpler way of finding the deposits greater than $80. Enter:

```
      (DEP>80)/DEP
85 125 89
```

selection expressions

The parentheses in the last expression ensure that *DEP*>80 is processed first so that the compression function (/) has the 0s and 1s it needs to work with. Expressions that appear to the left of the slash (such as *DEP*>80) are called selection expressions, because they generate the 0s and 1s that the system uses to select items in a vector or matrix.

You must use an equal number of numbers on either side of the slash. If you forget a 0 or 1, APL★PLUS displays a *LENGTH ERROR* message.

```
      1 0 0 0/DEP
LENGTH ERROR
      1 0 0 0/DEP
             ^
```

relational functions You can also use the following functions in selection expressions: less than (<), less than or equal to (≤), equal (=), greater than or equal to (≥), and not equal (≠). These are called relational functions, since they show a relationship between data. The symbols are usually located on the 3, 4, 5, 6, and 8 keys, respectively. Try the following expressions:

```
      DEP
85 125 60 45 89

      (DEP<60)/DEP
45

      (DEP≤60)/DEP
60 45

      (DEP=60)/DEP
60

      (DEP≥60)/DEP
85 125 60 89

      (DEP≠60)/DEP
85 125 45 89
```

Using Compression with Character Data

When you want to select certain elements from a list of numbers, you enter a selection expression (like >80), followed by a slash and the name of the variable that contains the list of numbers. You can also use selection expressions with character data, but only using the equal (=) and not equal (≠) symbols. Try entering:

```
      TEXT←'THIS SENTENCE HAS NO BLANKS.'
```

Now enter:

```
        ' '≠TEXT
```

The system displays a 1 for each nonblank character, and a 0 for blanks:

```
1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 1
1 1 1 1 1
```

Visually, this looks like:

```
    TEXT     THIS SENTENCE HAS NO BLANKS.
    ≠' '     1111011111111011101101111111
```

To display the nonblank characters you could use this string of 0s and 1s with the slash and variable name as you have done before. But an easier way to display the nonblank characters is:

```
        (' '≠TEXT)/TEXT
THISSENTENCEHASNOBLANKS.
```

In the preceding example, the `' '≠TEXT` portion of the expression produces 0s for blanks and 1s for everything else. The system uses the 1s to select and display all nonblank characters.

Try a few more examples. Enter the following expressions:

```
        ('PEAR'='BEAR')/'BEAR'
EAR

        ST←'***S**T**A**R**S***'
        (ST≠'*')/ST
STARS
```

If you do not understand a result that APL★PLUS displays, try entering the expression one piece at a time. For example, to analyze the second example, enter the part of the expression that is inside the parentheses.

```
        ST≠'*'
```

The system displays a string of 0s and 1s:

```
0 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 0
```

Then compare this string with the vector you are compressing. Enter:

```
      ST
***S**T**A**R**S***
```

The 1s correspond to the letters that the system displays when you execute the entire expression.

Remember, you cannot use relational functions other than equal (=) and not equal (≠) with character data.

```
      'PEAR'>'BEAR'
DOMAIN ERROR
      'PEAR'>'BEAR'
            ^
```

Counting Occurrences

There are many times you want to know how frequently a certain value or condition occurs in a group of data. For example, a teacher might want to know how many students achieved a perfect score on a given test, or a customer service representative might want to know how many customers called more than two times with the same problem.

Suppose you want to know how many times you deposited more than $80 in your bank account during the past year. The variable *DEP* lists each of your bank deposits during the period ($85, $125, $60, $45, and $89). You use the greater than symbol (>) to compare the deposits in the *DEP* variable to 80; that is:

```
      DEP>80
```

The system displays:

```
1 1 0 0 1
```

To find the number of deposits greater than $80, add the 1s in the list using the plus reduction function (+/). Enter:

```
      +/1 1 0 0 1
3
```

Or, more simply you can enter

```
      +/DEP>80
3
```

This approach works equally well with other relational functions.

```
      +/DEP=60      (deposits equal to $60)
1

      +/DEP≥20      (deposits greater than or equal to $20)
5
```

You can also use plus reduction (+/) with character data. To find out how times the letter I appears in MISSISSIPPI, enter:

```
      +/'I'='MISSISSIPPI'
4
```

Or, to find the number of blanks in a sentence, enter:

```
      S←'IT WAS THE BEST OF TIMES.'
      +/' '=S
5
```

Selecting Data Using Multiple Conditions

You can also select data using more than one condition. For example, instead of just finding deposits greater than $80, you might want to find deposits greater than $80 but less than $100.

And

The and function (∧) lets you to select data that meet two conditions. The symbol for the and function (∧) is located on the 0 key.

First, look at the deposits that are greater than $80. Then look at those that are less than $100. Enter:

```
      DEP>80
1 1 0 0 1
      DEP<100
1 0 1 1 1
```

Now use the and function (∧) to compare the 0s and 1s that result from the expressions *DEP*>80 and *DEP*<100. Where 1s occur in *both* lists (that is, where *both* conditions are met), the system displays a 1. Enter:

```
      1 1 0 0 1∧1 0 1 1 1
1 0 0 0 1
```

Visually, you can look at these statements as:

Deposit	$85	$125	$60	$45	$89
> $80 (1 = True)	1	1	0	0	1
<$100 (1 = True)	1	0	1	1	1
> $80 and <$100	1	0	0	0	1

Which deposits are greater than $80 and less than $100? Enter:

```
      1 0 0 0 1/DEP
85 89
```

Better yet, use the and function (∧) like this:

```
      ((DEP>80)∧DEP<100)/DEP
85 89
```

Or

The or function (∨) is similar to the and function. The or function (∨) chooses data that meet *either* of two requirements. The symbol for the or function (∨) is located on the 9 key.

This time, look at the deposits that are greater than $120. Then look at those that are less than $75. Enter:

```
      DEP>120
0 1 0 0 0

      DEP<75
0 0 1 1 0
```

Now use the or function (∨) to compare the 0s and 1s that result from the expressions *DEP*>120 and *DEP*<75. Where 1s occur in *either* list, the system displays a 1. Enter:

```
      0 1 0 0 0∨0 0 1 1 0
0 1 1 1 0
```

Visually, you can look at these statements as:

Deposit	$85	$125	$60	$45	$89
> $120 (1 = True)	0	1	0	0	0
<$75 (1 = True)	0	0	1	1	0
> $120 or <$75	0	1	1	1	0

Which deposits are greater than $120 or less than $75? Enter:

```
      0 1 1 1 0/DEP
125 60 45
```

An even easier way to use the or function (∨) to solve this problem is:

```
      ((DEP>120)∨DEP<75)/DEP
125 60 45
```

Checking All of the Data

You can use the and reduction function (∧/) and the or reduction function (∨/) to determine whether any or all of the data meet certain requirements. To form the and reduction function, combine the and function (∧) with a slash (/). To form the or reduction function, combine the or function (∨) with a slash (/).

and reduction

Imagine that your company has seven employees with salaries of $21,000, $18,000, $54,000, $24,000, $17,000, $20,000, and $29,500. Create the variable *SAL* (salary) by entering:

```
      SAL←21000 18000 54000 24000 17000
20000 29500
```

You want to know whether everyone in the company earns more than $12,000. Enter:

```
      SAL>12000
1 1 1 1 1 1 1

      ∧/1 1 1 1 1 1 1
1
```

The and reduction function (∧/) determines that the result is all 1s. So the answer is yes — everyone in the company earns more than $12,000.

However, a simpler statement is:

```
      ∧/SAL>12000
1
```

The *SAL*>12000 portion of the expression determines which elements of *SAL* are greater than 12000. The and reduction function (∧/) determines whether *all* of the salaries are greater than $12,000 (that is, whether the result of *SAL*>12000 is all 1s).

Does everyone on the company earn more than $17,000? Enter:

```
      SAL>17000
1 1 1 1 0 1 1
```

The system displays a 0 in the result because one of the salaries is not greater than $17,000. Combine this result with the and reduction function (∧/):

```
      ∧/1 1 1 1 0 1 1
0
```

Notice that one false result makes the entire result false. Now try a simpler statement:

```
      ∧/SAL>17000
0
```

or reduction

You can use the or reduction function (∨/) to find if *any* of the data meet a requirement. If one result is true, the entire result is true. For example, are any of the salaries in your company larger than $50,000? Enter:

```
      SAL
21000 18000 54000 24000 17000 20000 29500
      SAL>50000
0 0 1 0 0 0 0
      ∨/0 0 1 0 0 0 0
1
```

Or, using a simpler statement:

```
      ∨/SAL>50000
1
```

Are any of the salaries in your company larger than $60,000? Enter:

```
        ∨/SAL>60000
0
```

Selecting Data from Matrices

compressing columns

The compression function (/) works differently with matrices. Remember that a matrix has rows and columns. You must indicate whether you want to compress the rows or the columns.

First, create a variable called *MAT* that is a matrix with four rows and four columns. The variable contains the numbers from 1 to 16. Enter:

```
      MAT←4 4⍴⍳16
      MAT
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
```

Now you want to remove the first and third columns of the *MAT* variable. You use a list of 0s and 1s to indicate which columns to leave out and which to keep. Since you want to leave out the first and third columns and keep the second and fourth, you enter:

```
      0 1 0 1/MAT
 2  4
 6  8
10 12
14 16
```

compressing rows

To compress the *rows* of the *MAT* variable, you use ⌿ or include [1] in the expression. Try removing the first and third rows of *MAT*. Enter:

```
      0 1 0 1/[1]MAT
 5  6  7  8
13 14 15 16
```

Even easier is the statement:

```
      0 1 0 1⌿MAT
 5  6  7  8
13 14 15 16
```

You can also use the compression function (/) with character matrices. Try it. First create a character matrix with three rows and three columns as shown below. Enter:

```
      CHARMAT←3 3⍴'CATRATBAT'
      CHARMAT
CAT
RAT
BAT
```

Use the compression function (⌿) to display the third row of the *CHARMAT* variable:

```
      0 0 1⌿CHARMAT
BAT
```

Now try using the compression function (/) to display the third column of *CHARMAT*:

```
      0 0 1/CHARMAT
T
T
T
```

using length restrictions

The total number of 0s and 1s in the expression must equal the number of rows (or columns) in the matrix. For example, suppose you have a matrix with three rows and five columns. If you are compressing the three rows, you must use a total of three 0s and 1s. If you are compressing the five columns, you must use a total of five 0s and 1s. If the total number of 0s and 1s in the expression is not equal to the number of rows (or columns), the system displays an error message.

using vectors as keys

If you want to select elements of a matrix without knowing their positions, you can use a list of numbers as a key that allows you to search the matrix. Then, you can use the result of the search to compress the matrix.

For example, suppose there are four salesmen in your company whose employee ID numbers are 101, 112, 126, and 128. Create a variable *SALESMEN* that is a list of the employee ID numbers for these four salesmen. Enter:

```
      SALESMEN←101 112 126 128
```

During the first three months of this year, the salesmen's revenues were:

Salesman 101	$1075	$3023	$4010
Salesman 112	$2075	$750	$1189
Salesman 126	$2211	$3117	$2288
Salesman 128	$2779	$1077	$1928

Create a variable *SALES* that is a matrix of this sales data:

```
      SALES←4 3⍴1075 3023 4010 2075 750
1189 2211 3117 2288 2779 1077 1928

      SALES
 1075 3023 4010
 2075  750 1189
 2211 3117 2288
 2779 1077 1928
```

Notice that each row in the matrix corresponds to one salesman's revenues for the first three months of the year. Row 1 corresponds to Salesman 101, Row 2 corresponds to Salesman 112, and so on.

Where is Salesman 101 in the variable *SALESMEN*? Enter:

```
      SALESMEN=101
1 0 0 0
```

What are the revenue figures for Salesman 101? Use the list of 0s and 1s to select the row in *SALES* that contains the revenue figures. (Do not forget to use ⌿ in the expression to select the *row* that corresponds to Salesman 101.) Enter:

```
      1 0 0 0⌿SALES
1075 3023 4010
```

Or you can simply search the matrix using the variable *SALESMEN*:

```
      (SALESMEN=101)⌿SALES
1075  3023  4010
```

Remember, the total number of 0s and 1s in the expression must equal the number of rows (or columns) in the character matrix. If the total number of 0s and 1s in the expression is not equal to the number of rows (or columns), the system displays an error message.

Now try the same concept with a character matrix. Suppose that the four salesmen have the following street addresses:

Salesman 101	21 Elm Street
Salesman 112	5107 Main Street
Salesman 126	52 North Road
Salesman 128	1717 Iowa Avenue

Create a variable *ADDRESS* that is a character matrix of these addresses. (Remember to insert blanks between the addresses, as necessary, so that each address appears as a row.) Enter:

```
      ADDRESS←4 16⍴'21 ELM STREET   5107
MAIN STREET52 NORTH ROAD   1717 IOWA
AVENUE'

      ADDRESS
21 ELM STREET
5107 MAIN STREET
52 NORTH ROAD
1717 IOWA AVENUE
```

What is the address of Salesman 101? Enter:

```
      (SALESMEN=101)⌿ADDRESS
21 ELM STREET
```

Now that you know how to use a list of numbers as a key to compress a matrix, you can try combining expressions.

Remember that in the *SALES* variable, the columns represent sales in the first, second, and third months of this year:

```
      SALES
 1075 3023 4010
 2075  750 1189
 2211 3117 2288
 2779 1077 1928
```

Create a variable called *MONTH* to store the numbers of the months. Enter:

```
      MONTH←1 2 3
```

Now you can select the sales data for the third month by entering:

```
      (MONTH=3)/SALES
 4010
 1189
 2288
 1928
```

As you learned earlier, you can select the row in *SALES* that contains Salesman 101's sales by entering:

```
      (SALESMEN=101)⌿SALES
1075 3023 4010
```

By combining these two expressions, you can now find the sales for Salesman 101 during Month 3:

```
      (MONTH=3)/(SALESMEN=101)⌿SALES
4010
```

Note: Be sure to save your workspace before ending this chapter and beginning the exercises. Enter:

```
      )SAVE MYWORK
```

Summary

In this chapter you learned that:

- You can use variations of the expression (DEP>80)/DEP to select data that meet certain requirements. Expressions of this form combine a relational function — less than (<), less than or equal to (≤), equal (=), greater than (>), greater than or equal to (≥), not equal (≠) — with the compression function (/).

- You can use variations of the expression +/DEP>80 to determine how many items meet a certain requirement.

- You can use variations of the expression ((DEP>80)∧DEP<100)/DEP to select data that meet *two* requirements. This expression uses the and function (∧).

- You can use variations of the expression ((DEP>120)∨DEP<75)/DEP to select data that meet *either* of two conditions. This expression uses the or function (∨).

- You can use variations of the expression ∧/SAL>17000 to determine if *all* of the data meet a certain requirement. This expression uses the and reduction function (∧/).

- You can use variations of the expression ∨/SAL>50000 to determine if *any* of the data meet a certain requirement. This expression uses the or reduction function (∨/).

- You can use vectors as keys to select data from matrices or from other vectors, using the selection expressions described above. Use ⌿ to select from the rows and use / to select from the columns.

Exercises

A. 1. Write an APL expression that counts the number of times that the letter E appears in TENNESSEE.

2. Create the *TRANSACT* variable shown below. This variable contains a vector of bank transactions *TRANSACT*.

```
      TRANSACT←180 ¯25.25 ¯40.89
87.12 237.25 ¯127.27 39.45
```

a. Write an expression that counts the number of withdrawals (negative numbers).

b. Write an expression that counts the number of deposits (positive numbers).

3. Write an APL expression that counts the number of blanks in the following sentence:

```
      S←'TO BE OR NOT TO BE, THAT IS
THE QUESTION'
```

4. Create the following vector of sales figures:

```
      SALES←757 212 358 821 375 115
837 227 706
```

Write an expression that counts the number of sales that are less than or equal to $375.

B. Create the *ACCOUNTS* variable shown below. This variable contains the account balances of ten bank customers.

```
      ACCOUNTS←135 27 ¯15 12 ¯25 ¯57 240
172 18 29
```

Write APL expressions to answer the following questions.

1. Do all of the accounts have *positive* balances? How many?

2. How many accounts have negative balances?

3. Are there any accounts with balances greater than $200? How many?

4. Are there any accounts with balances between $120 and $175? How many?

5. What is the grand total of the accounts with negative balances? (Hint: Use compression to find the negative balances, then add them using plus reduction.)

7. What is the grand total of the accounts with positive balances?

C. Use the following price information to solve the exercises in this section:

```
      PRICES←1.79 5.21 10.77 30.29 18.95
19.95 3.07 2.15
```

1. Write an expression that selects the prices larger than $15.00.

2. Write an expression that selects the prices that are less than or equal to $3.07.

3. Write an expression that selects any prices that are equal to $17.75.

4. Write an expression that selects prices that are larger than $10 but smaller than $20.

5. Write an expression that selects numbers that are either less than $2 or greater than $30.

D. Use the following variables to solve the exercises in this section:

```
      A←3 3ρι9
      A
1 2 3
4 5 6
7 8 9
      CM←3 3ρ'MOEJOETOE'
      CM
MOE
JOE
TOE
```

1. `0 0 1/A`

2. `0 1 1/CM`

3. `1 0 0⌿CM`

4. `0 1 0⌿A`

5. `0 0 1/0 1 0⌿A`

6. `0 0 0 1/A`

E. Use the following variables to solve the exercises in this section:

```
      CUSTNO←1 2 3 4
      MONTHS←10 11 12
      CADDRESS←4 16ρ'55 WESTLAKE DR. 101
JEFFREY LN. 23 REPUBLIC AVE.27 MEMORIAL
DR. '
      CORDERS←4 3ρ73 412 811 27 84 11 72
35 99 107 78 23
```

Variables *CUSTNO* and *MONTHS* are keys to the other data. The customer numbers (*CUSTNO*) correspond to the *rows* of *CADDRESS* and *CORDERS*. The months correspond to the *columns* of *CORDERS* (that is, *CORDERS* contains customer orders for three months: October, November, December).

1. Write an expression that displays the orders for Customer 3 during this period.

2. Write an expression to display the address for Customer 4.

3. Write an expression that displays all the orders for the month of November.

4. Write an expression that displays the orders for Customer 1 during the month of December.

4

Exploring APL

In this chapter, you will learn about APL functions that sort data, manipulate data in a variety of ways, and perform math functions — from rounding numbers to solving simultaneous equations.

Unlike previous chapters, this chapter does not include specific exercises. It does, however, present some APL functions that you will see in the APL programs in later chapters — useful programming tools that you can incorporate into your own APL programs.

Sorting

grade up and grade down

You use the APL functions grade up (⍋) and grade down (⍒) to sort data. However, the grade up and grade down functions do not themselves sort the data. Rather, these functions return the indices necessary to sort the data in ascending or descending order. To actually sort the data, you use the index function ([]) with the grade up and grade down functions.

To see how the grade up (⍋) and grade down (⍒) functions work, first create a variable *DATA* that contains the values 62, 80, 47, and 24:

```
        DATA←62 80 47 24
```

producing indices to sort data

To produce the indices you need to sort *DATA* in ascending order, enter:

```
        ⍋DATA
```

The system displays:

```
4 3 1 2
```

using grade up and grade down with index

The result of the grade up function (⍋) indicates that the fourth element (24) is the smallest number, the third element (47) is the next smallest, and so on. To sort the *DATA* variable in ascending order, you use the index function ([]) with the grade up function (⍋). Try it. Enter:

```
        DATA[⍋DATA]
```

APL★PLUS executes ⍋*DATA* first, then uses the result as the index to *DATA*. The system then displays the result of the index operation — the data sorted in ascending order:

```
24 47 62 80
```

To see how the system generates indices with the grade down function (⍒), enter:

```
      ⍒DATA
```

The system displays:

```
2 1 3 4
```

Now sort *DATA* in descending order. Enter:

```
      DATA[⍒DATA]
```

The system displays:

```
80 62 47 24
```

sorting identical values

When there are two identical values in the data, the grade up (⍋) and grade down (⍒) functions give the left value the lower index. For example, if you enter:

```
      DATA2←62 80 47 24 80
      ⍋DATA2
```

The system displays:

```
4 3 1 2 5
```

Grade up gives the first occurrence of 80 (the leftmost) the lower of the two indices, and the second occurrence of 80 the higher of the two indices.

To sort and display *DATA2* in ascending order, enter the line with the six-space indent that is shown below. The system response begins at the left margin.

```
      DATA2[⍋DATA2]
24 47 62 80 80
```

When the grade down function (⍒) encounters two identical values, it gives the first occurrence of 80 the lower of the two indices, and the second occurrence the higher of the two indices; for example:

```
      ⍒DATA2
2 5 1 3 4
```

To sort and display *DATA2* in descending order, enter:

```
      DATA2[⍒DATA2]
80 80 62 47 24
```

sorting matrices

When you sort a matrix, the grade up (⍋) and grade down (⍒) functions sort along the rows. They treat the first column of the table as a vector and sort it, moving to the second column only if there are duplicates in the first column. Unlike sorting with vectors, using the grade up (⍋) and grade down (⍒) functions with matrices does not give the left value the lower index.

To see how the grade functions sort matrices, first create and display a matrix called *DATA3*. The variable has four rows and two columns and contains the values 54, 100, 22, 49, 22, 23, 62, and 99. Enter:

```
      DATA3←4 2⍴54 100 22 49 22 23 62 99
      DATA3
54  100
22   49
22   23
62   99
```

When you use the grade up function (⍋) with *DATA3*, the system produces the indices you need to sort the rows of the matrix:

```
      ⍋DATA3
3 2 1 4
```

Because the first column contains duplicate values, the grade up function (⍋) sorts the corresponding values in the second column to break the tie. Because 23 is smaller than 49, the third row comes before (has a lower index than) the second row.

To sort the matrix in ascending order, you use the index function ([]) with the grade up function (⍋) — just as you do when sorting the elements in a vector. However, when you are sorting a matrix, you must tell the system whether you want to sort the rows or the columns. To index into the *rows* of *DATA*3, you include a semicolon before the right bracket (]). Therefore, to produce the sorted matrix in ascending row order, you enter:

```
      DATA3[⍋DATA3;]
```

The result is:

```
22   23
22   49
54  100
62   99
```

You know that the grade up (⍋) and grade down (⍒) functions sort matrices by sorting the first column, then the second, and so on. But you can change this default order by specifying the relative importance, or significance, of the columns.

For example, suppose you want to sort a group of people by age, from oldest to youngest. You can specify their birth dates as a matrix, where each contains a birth date by month, day, and year. First, create a matrix called *AGES* with four rows and three columns that contains the birthdates 11/25/46, 1/18/52, 11/21/36, and 7/23/55:

```
      AGES←4 3⍴11 25 46 1 18 52 11 21 36 7
23 55
      AGES
11 25 46
 1 18 52
11 21 36
 7 23 55
```

Because the third column contains the birth year, it is the most significant. The first column, the birth month, is the next most significant. The second column, the birth day, is the least significant.

To index into the *columns* of a matrix, you include a semicolon after the left bracket ([). To produce the indices you need to sort *AGES* in ascending order based on the significance of the columns, you enter:

```
        ⍋AGES[;3 1 2]
```

The grade up function (⍋) sorts the third column first. Because the third column has no duplicate values, the grade up function stops there. If the third column did contain duplicate values, the grade up function would then sort the first column. If the first column contained duplicates, the grade up function would then sort the second column.

The system displays the row indices that you need to sort the matrix:

```
3 1 2 4
```

The third row contains the birth date of the oldest person, the first row contains the birth date of the second oldest, and so on.

Now, to sort the matrix, enter:

```
        AGES[⍋AGES[;3 1 2];]
```

The system displays:

```
11 21 36
11 25 46
 1 18 52
 7 23 55
```

sorting character data

To sort character data, you must use a sorting sequence on the left side of ⍋ or ⍒. The sorting sequence specifies the order of the characters. A convenient sorting sequence is ⎕*AV*, the atomic vector, which contains all possible characters used in APL★PLUS.

For example, imagine that you want to sort the names Jim, June, John, and Joan into alphabetical order. First, create a four-by-four matrix (a matrix with four rows and four columns) of the names. Enter:

```
      NAMES←4 4⍴'JIM JUNEJOHNJOAN'
      NAMES
JIM
JUNE
JOHN
JOAN
```

Then use ⎕*AV* on the left side of the grade up function (⍋) to produce the indices you need to sort the matrix *NAMES*. Enter:

```
      ⎕AV⍋NAMES
1 4 3 2
```

The indices show that Row 1 is first in alphabetical order, Row 4 is second, Row 3 is third, and Row 2 is fourth. To sort the matrix, enter:

```
      NAMES[⎕AV⍋NAMES;]
JIM
JOAN
JOHN
JUNE
```

Instead of using ⎕*AV*, you can use a vector that contains the alphabet , as shown in the following example.

```
      ALPH←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      NAMES[ALPH⍋NAMES;]
JIM
JOAN
JOHN
JUNE
```

Rounding

In an earlier chapter, you used the APL functions minimum (⌊) and maximum (⌈) with the reduction operator (/) to find the smallest (lowest) and largest (highest) values in a vector. For example, if you enter

```
      ⌊/10 4 7 15
```

the system finds the smallest number and displays:

```
4
```

If you enter

```
      ⌈/10 4 7 15
```

the system finds the largest number and displays:

```
15
```

You can also use these functions monadically to compute either the next lowest whole number (minimum) or the next highest whole number (maximum). In the following examples, you enter the lines that begin with a six-space indent; the system displays the lines at the margin.

```
      ⌊4.1 4.9 ¯1.2
4 4 ¯2

      ⌈4.1 4.9 ¯1.2
5 5 ¯1
```

As you can see, the minimum (⌊) and maximum (⌈) functions do not actually round numbers. When you round numbers, you want 4.1 to round down to 4, and 4.9 to round up to 5; that is, round to the *nearest* whole number. To round numbers, use the following formula with minimum and maximum:

```
      ⌊.5 + 4.1 4.9 ¯1.2
```

The system displays:

```
4 5 ¯1
```

Adding .5 to 4.9 results in 5.4, which correctly rounds to 5.

Converting Characters to Numbers

You have already learned how to convert numbers to characters using the format function (⍕). You can also convert characters to numbers with the execute function (⍎). You use Shift-; to create the ⍎ symbol.

execute function

The execute function uses the ⍎ symbol. In the first example, the execute function converts the characters 59 72 63 to their numeric equivalents:

```
      ⍎ '59 72 63'
59 72 63
```

Although 59, 72, and 63 look like numbers, they are characters because they are enclosed in single quotes. If you try to add 5 to each element in the vector, the system displays an error:

```
      5 + '59 72 63'
DOMAIN ERROR
      5 + '59 72 63'
        ^
```

If you use the execute function (⍎) to change the characters to numbers, you can then add 5 to them; for example:

```
      5 + ⍎ '59 72 63'
64 77 68
```

You can use the execute function (⍎) with any list of characters that looks like a valid APL expression. For example, try the following expression:

```
      ⍎ 'MAT←2 3⍴⍳6'
      MAT
1 2 3
4 5 6
```

If the list of characters contains something other than numbers and valid variable or function names, the system displays an error.

```
        ⍎ '35 92 HI THERE'
SYNTAX ERROR
        ⍎ 35 92 HI THERE
                       ^
```

format inverse function

In the previous example, you cannot use the format function (⍎). However, you can use the APL★PLUS system function ⎕*FI* (format inverse). ⎕*FI* converts valid numbers from character data to numeric data, and changes invalid numbers to 0. In the example, 35 and 92 are valid numbers, but the words *HI* and *THERE* are not.

```
        ⎕FI '35 92 HI THERE'
35 92 0 0
```

verify input function

The system function ⎕*VI* (verify input) tells you what a character string contains. The system displays 1s for valid numbers and 0s for everything else; for example:

```
        ⎕VI '12.1 HI 52 ⍉⍋'
1 0 1 0
```

Therefore, to extract the real numbers from a text string, you use the compression function (/):

```
        (⎕VI TEXT)/⎕FI TEXT
```

Scalar, Vector, or Matrix?

Sometimes you need to know whether your data are scalars, vectors, or matrices. For example, some functions (such as count) only work with scalars, and other functions (such as index) work with everything except scalars.

Rank

APL uses the concept of rank to meet this need. Rank:defined, is the number of dimensions of the data. The function for finding rank is ρρ — the shape of the shape.

Remember that the shape of a vector is a single number showing how many elements the vector contains; for example:

```
      ρ5 8 10
3
```

If you find the shape of the shape (ρρ) of this vector, you get the rank:

```
      ρρ5 8 10
1
```

The shape of a matrix has two numbers — one for the rows and one for the columns; for example:

```
      MAT←2 4ρ3 2 1 0
      MAT
3 2 1 0
3 2 1 0
      ρMAT
2 4
```

If you find the shape of the shape (ρ ρ) of this matrix, you get the rank. Because there are two elements in the shape, the rank of a matrix is 2:

```
      ρρMAT
2
```

This technique also works with character data; for example:

```
      ρρ10 20ρ'HELLO THERE'
2
```

A scalar has no shape — a shape of 0. (The exact definition of a scalar's shape is the empty vector ι 0 or ⍬.) For example, at the six-space indent enter:

```
      ρ5
```

The system moves the cursor down to the next line and indents the cursor six spaces without displaying anything.

So a scalar's rank is 0:

```
      ρρ5
0
      ρι0
0
```

However, since an empty vector is still a vector, it has a rank of 1.

```
      ρρι0
1
```

Understanding rank becomes simple when you think of it in terms of basic geometry. A scalar corresponds to a point; a vector corresponds to a line; and a matrix corresponds to a plane. Like a point, a scalar has a position but no length or depth. Line a line, a vector has length but no depth. Like a plane, a matrix has both length and depth. Therefore, a scalar has zero dimensions, a vector has one dimension, and a matrix has two dimensions.

Multi-Dimensional Objects

APL also allows data with higher ranks. You call data with more than two dimensions multi-dimensional objects.

The system allows you to deal with multi-dimensional objects as easily as scalars. You can build multi-dimensional objects with the reshape function, much as you build matrices. The following example shows a three-dimensional matrix with two planes, four rows, and four columns.

```
      THREED←2 4 4⍴1 2 3 4
      THREED
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4

1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
```

APL represents multi-dimensional objects by sets of tables and displays them by separating the planes with a blank line.

It is hard to visualize objects with five or six dimensions. But, you can always use rank to determine the number of dimensions; for example:

```
      ⍴⍴THREED
3

      ⍴⍴3 4 5 2 4⍴0 1 2 3
5
```

Manipulating Data

This section presents a number of functions you can use to manipulate data. You will learn such techniques as changing scalars and matrices to vectors, altering the position of data elements, reversing the order of data elements, and so on.

Ravel

changing a scalar or matrix to a vector

You may want to convert data to a vector, because a vector can be easier to work with than a matrix or a scalar. The ravel function does this. The ravel function uses the comma, as does catenate; however, catenate uses the comma with data on both sides (dyadically) while ravel uses the comma with data only on the right side (monadically).

To see how the ravel function (,) works, first create and display the following matrix:

```
      MAT←2 3⍴16+⍳6
      MAT
17 18 19
20 21 22
```

To change *MAT* from a matrix to a vector, enter:

```
      ,MAT
```

The system displays:

```
17 18 19 20 21 22
```

You can see where the ravel function gets its name. Like pulling a thread from a piece of cloth, the ravel function "ravels out" the matrix — pulling successive rows of the matrix into a long string of numbers. Ravel also turns a scalar into a one-element vector; for example:

```
      ρ5
(six-space indent)
      ,5
5
      ρ,5
1
```

As the preceding example shows, a scalar has no shape, but a raveled scalar has a shape of 1.

The ravel function (,) is useful when you want to ensure that the data you are using is a vector and not a scalar or matrix. For example, if you are going to index into the data, you must convert scalars to vectors, because indexing is not defined for scalars.

Rotate

The rotate function does exactly what you would expect it to — it rotates the data. The rotate function uses the ⌽ symbol.

changing the order of data

When you use the rotate function (⌽) monadically (with data on one side), it reverses the order of the data completely; for example:

```
      ⌽⍳4
4 3 2 1
```

With a matrix, the rotate function reverses the order of the columns within a row; for example:

```
      MAT←3 4 ρ'ABCDEFGHIJKL'
      MAT
ABCD
EFGH
IJKL

      ⌽MAT
DCBA
HGFE
LKJI
```

specifying the order

You can also use the rotate function dyadically (with data on both sides). The left argument specifies how to rotate the data. For example, to rotate a vector two positions to the left, enter:

```
      2⌽4 5 6 7 8
```

The system displays:

```
6 7 8 4 5
```

To rotate data in the opposite direction, use a negative number in the left argument :

```
      ¯1⌽4 5 6 7 8
8 4 5 6 7
```

When you use the dyadic rotate function with a matrix, the left argument specifies how to rotate the individual columns or rows. For example, create and display the following matrix.

```
      MAT←3 4⍴'ABCDEFGHIJKL'
      MAT
ABCD
EFGH
IJKL
```

To rotate the first row by 1, the second by 2, and the third by 1, specify the left argument as a three-element vector:

```
      1 2 1⌽MAT
BCDA
GHEF
JKLI
```

To rotate columns instead of rows, enter:

```
       2 3 2 ¯1⌽[1]MAT
IBKL
AFCD
EJGH
```

In this example, you rotated the second column by 3, which brought it back to its starting position.

overturn

You can use the overturn function as a synonym for ⌽[1]. The overturn function uses the ⊖ symbol. When you use the overturn function monadically, it reverses the order of the rows completely:

```
      ⊖MAT
IJKL
EFGH
ABCD
```

You can also use the overturn function (⊖) dyadically. Similar to the rotate function, the left argument of the overturn function specifies how you want to rotate each column. The next example rotates the first column by 2, the second column by 3, the third column by 2, and the fourth column by ¯1.

```
      2 3 2 ¯1⊖MAT
IBKL
AFCD
EJGH
```

Transpose

The transpose function (⍉) allows you to shift the dimensions of your data. For example, you can change the rows dimension into the columns dimension:

```
      MAT
ABCD
EFGH
IJKL
      ⍴MAT
3 4
      ⍉MAT
AEI
BFJ
CGK
DHL
      ⍴ ⍉MAT
4 3
```

You can also use the transpose function (⍉) dyadically. The left argument specifies how to change the dimensions. The next example tells the system to make the first dimension the second, and vice versa. So, the rows (such as *ABCD*) are now columns.

```
      2 1⍉MAT
AEI
BFJ
CGK
DHL
```

The transpose function (⍉) is useful when you want to sort the columns of a matrix rather than the rows. (Remember that the grade up (⍋) and grade down (⍒) functions sort the rows of a matrix.)

Take

The take function, which uses the ↑ symbol, does exactly what its name says — it takes as many elements from the right argument as you specify in the left argument. You can use the take function (↑) with a scalar, vector, or matrix. The following examples show the take function used with vectors. In the first example, the take function takes the first four elements of *VECT*, because the left argument is 4. In the second example, the take function takes the first nine elements of *S* — including a space. (Remember that a space is a character.)

```
      VECT←⍳10
      4↑VECT
1 2 3 4

      S←'ONCE UPON A TIME'
      9↑S
ONCE UPON
```

If the left argument is a negative number, the system takes the elements from the end of the vector; for example:

```
      ¯3↑VECT
8 9 10

      ¯4↑S
TIME
```

If you specify more elements than the vector has, the take function pads the vector with zeros (for a numeric vector) or blanks (for a character vector). The next three examples show this behavior.

```
      12↑VECT
1 2 3 4 5 6 7 8 9 10 0 0

      ¯12↑VECT
0 0 1 2 3 4 5 6 7 8 9 10

      ¯15↑'OH HO'
          OH HO
      ⍴¯15↑'OH HO'
15
```

To apply the take function (↑) to a matrix, you need two elements in the left argument. The first element specifies the number of rows to take; the second specifies the number of columns to take; for example:

```
      2 5 ↑ 3 4 ⍴ ⍳12
1 2 3 4 0
5 6 7 8 0
```

Drop

The drop function, which uses the ↓ symbol, does the opposite of the take function. It drops as much data as you specify. For example, to drop the first two items in a five item vector, enter:

```
      2↓⍳5
```

The system displays:

```
3 4 5
```

Use negative numbers to drop items from the end of a vector. For example, to drop the last two items from the vector, enter:

```
      ¯2↓ι5
```

The system displays:

```
1 2 3
```

Using the drop function with matrices can be confusing, because it drops both rows and columns. For example, try dropping the first two rows and the first column in *MAT*:

```
      MAT
ABCD
EFGH
IJKL
      2 1↓MAT
```

The system displays:

```
JKL
```

To eliminate only the first row of *MAT*, and leave all columns, enter:

```
      1 0↓MAT
```

The system displays:

```
EFGH
IJKL
```

It can be very useful to use the take (↑) and drop (↓) functions together to move through your data, processing individual chunks of it. For example, create the following vector:

```
      DATA←'PIECE1PIECE2PIECE3'
```

To select the first chunk of data, use the take function (↑) to select the first six characters. Enter:

```
      6↑DATA
```

The system displays:

```
PIECE1
```

To select the next chunk of data, use the drop function (↓) to drop the first six characters and then use the take function (↑) to select the next six characters:

```
      6↑6↓DATA
```

The system displays:

```
PIECE2
```

Locating Data

At times, you may need to know if something exists ("Did we have any sales to customer 515 last month?"). If that data does exist, you may want to know where it is ("Which salesman made that sale?"). You looked at one way of answering these questions in the previous chapter. This section examines some other ways of locating data.

Membership

When you want to know whether something exists or not, you can use the membership function (∊). This function is also called element of. The membership function determines whether the number (or character) to the left of the ∊ (the left argument) exists in the numbers (or characters) to the right of the ∊ (the right argument) The answer is either true or false. If the answer is true, the membership function returns a 1; if the answer is false, the membership function returns a 0.

For example, to determine whether 5 is an element of the series 1 through 6 you enter:

```
      5 ∊ 1 2 3 4 5 6
```

Because 5 is an element of the series, the system displays:

```
1
```

To determine whether 5 is an element of the vector 0 6 99, you enter:

```
      5 ∊ 0 6 99
```

Because 5 is not part of the vector, the system displays:

```
0
```

The left and right arguments to the membership function (∊) can have different lengths. For example, to determine whether the numbers in the vector 5 6 11 are all members of the vector generated by ι 10, you enter:

```
          5 6 11 ∊ ι10
```

(Remember that ι 10 produces a list of the numbers from 1 to 10.) The system displays 1s for the 5 and 6, and a 0 for the 11:

```
1 1 0
```

You can also use the membership function (∊) with character data. For example, to determine if the letter A is part of the word MARY, enter:

```
          'A' ∊ 'MARY'
```

The system displays:

```
1
```

In addition, you can determine if all values of one object are contained in another object by using an expression such as:

```
          ∧/DATA1 ∊ DATA2
```

The system displays a 1 if the left argument is a subset of the right argument; otherwise, the system displays a 0. For example, enter:

```
          ∧/1 9 8 ∊ ι10
```

The system displays

```
1
```

because 1, 9, and 8 are all members of the vector produced by ι 10.

Where

Once you know that something exists, you may need to know where it is located in the data. You can use ι dyadically (with data to the left and right of the ι) to do this. This use of ι is called the where function. Given a number (or character) to the right of the ι and a list of numbers (or characters) to the left of the ι, the where function (ι) displays the position of that number (or character) in the list of numbers. In other words, the where function displays the index of the specified number in the list of numbers. For example, enter:

```
      2 6 5 9ι5
```

The system displays

```
3
```

because 5 is in the third position of the left argument. To check this answer, use indexing to find the third element of the vector:

```
      2 6 5 9[3]
5
```

In the first example, 5 is located in the third position of the list of numbers.

If the number that you specify to the right of the ι does not exist in the list of numbers, the where function (ι) returns the index of the last element in the list plus 1. Enter:

```
      2 6 5 9ι44
```

The system displays:

```
5
```

In this example, 44 is not in the list of numbers, so the system displays the index of the last element (4) plus 1; that is, 5.

Outer Product

The outer product function uses an APL operator, ∘. (pronounced "jot dot" or "null dot"), to combine each element of the left argument with each element of the right argument. For instance, an easy way to produce a multiplication table is:

```
      1 2 3∘.×1 2 3 4
  1  2  3   4
  2  4  6   8
  3  6  9  12
```

The first row of the result shows 1 times 1 2 3 4, the second row shows 2 times 1 2 3 4, and so on.

You can use any scalar dyadic function (any arithmetic, relational, or logical function that uses two arguments) with outer product. Some of the scalar dyadic functions you can use are:

Arithmetic

add (+)
subtract (-)
multiply (×)
divide (÷)
maximum (⌈)
minimum (⌊)
power (*)

Relational

less than (<)
less than or equal (≤)
equal (=)
greater than or equal (≥)
greater than (>)
not equal (≠)

Logical

and (∧)
or (∨)

The next example uses the not equal function (≠). Each element in the left argument is compared to each element in the right argument. If the two numbers are not equal, the result is 1; otherwise, the result is 0.

```
      1 2 3∘.≠1 2 3
0 1 1
1 0 1
1 1 0
```

The next example uses the greater than or equal to function (≥) instead of not equal.

```
      1 2 3∘.≥1 2 3
1 0 0
1 1 0
1 1 1
```

Inner Product

Inner product is an even more powerful operation. Inner product takes two scalar dyadic functions as its arguments, one on each side of the dot (.). It uses the function on the right to combine each element of the left argument with each element of the right argument (similar to outer product). After completing that operation, it uses the function on the left to reduce the result.

One example of inner product is the table lookup function (∧.=) that you learned about earlier in this chapter.

matrix multiplication

Another example of inner product is to use + and × to perform matrix multiplication. First, create and display the following matrix:

```
      M1←2 3⍴⍳6
      M1
1 2 3
4 5 6
```

Then create and display a second matrix by transposing the columns ands rows in *M*1:

```
      M2←⍉M1
      M2
1 4
2 5
3 6
```

To multiply the matrices together, use the inner product function `+.×`; for example:

```
      M1+.×M2
14 32
32 77
```

This function multiplies each row of the left argument by each column of the right argument, then uses plus reduction on the results. Notice that the left argument must have the same number of columns as the right argument has rows. The result then has the same number of rows as *M*1, but the same number of columns as *M*2.

You can use any of the scalar dyadic functions as either the left or right functions in inner product.

Matrix Division

APL also provides a function to solve matrix division problems. Given a matrix *M* and a vector *R* (with the same number of elements as *M* has rows), the expression *R*⌹*M* solves the equation MX=R, where X is a vector of unknowns. The matrix divide function uses the ⌹ symbol. The following example shows matrix division.

```
      M←3 3ρ1 1 1 0 1 1 0 0 1
      M
1 1 1
0 1 1
0 0 1
      2 3 5⌹M
¯1 ¯2 5
```

If the equation has more than one solution, or if matrix *M* is singular (cannot be inverted), ⌹ returns a *DOMAIN ERROR*.

Least-Squares Approximation

If *M* has more rows than columns, *R*⌹*M* provides a least-squares approximation.

```
        M2←3 2 ⍴ 1 3 4 2 1 1
        14 26 7 ⌹ M2
4.981481481 2.944444444
```

You can use dyadic ⌹ to solve linear equations, to fit a straight line, and to fit a polynomial curve. For example, consider the following set of three linear equations:

$x + y + 2z = 9$
$2x + 4y - 3z = 1$
$3x + 6y - 5z = 0$

You represent these equations in APL with two matrices

```
        A←3 3 ⍴ 1 1 2 2 4 ¯3 3 6 ¯5
        A
   1  1  2
   2  4 ¯3
   3  6 ¯5

        B←3 1 ⍴ 9 1 0
        B
9
1
0
```

where *A* represents the coefficients of the linear equations and *B* represents the resultants.

To determine the solutions for x, y, and z in the linear equations, enter:

```
        B⌹A
1
2
3
```

Therefore, the solution is x = 1, y = 2, and z = 3.

Matrix Inversion

When you use ⌹ monadically, it returns the inverse of matrix *M*, provided that *M* is square and nonsingular. If *M* has more rows than columns, the system returns a least-squares approximation to the right inverse of *M*.

```
      ⌹2 2⍴1 1 0 1
1 ¯1
0  1
```

Combinations

Use the combinations function to find the number of distinct groups of a particular size that can be selected from a group of equal or larger size. Such problems are usually described as "five choose two," "four choose three," and so on. The combinations function uses the "shriek" symbol, !, as shown below.

```
      3!6
20
      0 1 2 3!3
1 3 3 1
```

Trigonometric Functions

APL provides a full set of functions to find values for trigonometric analysis. All are based around the circular function, which uses the ○ symbol. When you use ○ monadically, it is defined as "pi times x," as shown in the next example.

```
      ○1
3.141592654
```

You use the circular function dyadically with a set of numeric codes as its left argument, ranging from 1 to 7 and from ¯1 to ¯7. For example, 1○*X* returns the sine of *X*, ¯1○*X* returns the cosine, 3○*X* returns the tangent, ¯3○*X* the arctangent, and so on. *X* is given in radians, which can be found using the formula:

```
      ○ DEGREES ÷ 180.
```

See your system reference manual for a complete list of the trigonometric functions.

Using APL Idioms

Even a language like APL, with its large set of built-in functions, does not include a function for every need. However, you can put together a few APL functions and form short expressions that are so generally useful that they almost serve as primitive functions themselves. These short expressions are called APL idioms.

Once you have used an idiom for a while, you learn to recognize it immediately, without having to decipher its individual parts. If you know a lot of common idioms, you can easily read other APL programs.

You have already learned a few idioms in the earlier chapters of this book; for example,

```
        (SENTENCE≠' ')/SENTENCE
```

is the idiom for removing all blanks in a character vector. The expression for rank,

```
        ⍴⍴DATA
```

is also an idiom. In fact, most of the expressions presented in this chapter can be considered idioms. This section introduces a few more useful idioms. More idioms can be found in any standard APL text.

Numeric or Character?

To find out whether your data are numeric or not, you use the idiom:

```
        0=1↑0⍴DATA
```

This idiom returns 1 for numeric data and 0 for character data; for example:

```
      0=1↑0⍴88 99 32
1
      0=1↑0⍴'HELLO THERE'
0
```

The corresponding test for character data is:

```
      ' '=1↑0⍴DATA
```

Finding Indices

A common idiom for finding all the indices of a vector is

```
      ⍳⍴VECT
```

To see how this idiom works, create and display the following vector:

```
      V←8 17 ¯2
```

Use ⍴ to determine the shape.

```
      ⍴V
3
```

Because the shape of *V* is 3, you can use ⍳ to generate the indices for *V*:

```
      ⍳⍴V
1 2 3
```

Now, you can use the idiom to display the contents of *V*:

```
      V[⍳⍴V]
8 17 ¯2
```

Using a Test to Locate Data

You can use the idiom (⍳⍴*VECT*) to form longer idioms. For example, the previous example shows that you get all of *V* if you index it with ⍳⍴*V*. But you can index *V* based on a test. The following expression generates the indices of all elements in *V* that are greater than zero:

```
      (V>0)/⍳⍴V
1 2
```

Once you have the indices, you can index *V* and locate all the numbers greater than zero:

```
      V[(V>0)/⍳⍴V]
8 17
```

Replacing Data in a Vector

A related idiom allows you to replace all occurrences of an item in vector with a new item. For example, to replace all of the 9s in vector *V* with 777, enter:

```
      V←8 9 12 27 9
      V[(V=9)/⍳⍴V]←777
```

This expression uses the compression function (/) inside the brackets to find the 9s. The *V*=9 portion of the statement inside the brackets generates a vector of 1s and 0s. The ⍳⍴*V* portion of the statement generates the indices. After the system evaluates these expressions, the statement looks like:

```
      V[0 1 0 0 1/1 2 3 4 5]←777
```

The compression function then uses the 1s and 0s to select the indices of the 9s. After the system evaluates the compression function, the statement looks like:

```
      V[2 5]←777
```

This statement assigns the value 777 to elements 2 and 5 of *V*, replacing the 9s that were in those positions. Display *V* to check:

```
      V
8 777 12 27 777
```

You can also use this idiom to replace character data in a vector. The next example replaces each S with an F:

```
      V←'MISSISSIPPI'
      V[(V='S')/⍳⍴V]←'F'
      V
MIFFIFFIPPI
```

Finding a Substring

The final idiom to be introduced here is not an idiom in the APL★PLUS System, but rather the system function ⎕*SS* (for string search). It appears here because it replaces long and complex idioms in other versions of APL.

To find a substring in a sentence, use ⎕*SS*. The left argument is the text you want to search, and the right argument is the substring you want to find.

APL displays a string of 0s and 1s , with a 1 marking the beginning of the substring.

```
      TEXT←'I YAM WHAT I YAM.'
      TEXT ⎕SS 'YAM'

0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0
```

If you want to know whether the string is in the text or not, enter:

```
      1∊TEXT ⎕SS 'YAM'
1
```

To determine where the string starts in the text, enter:

```
      (TEXT ⎕SS 'YAM')/⍳⍴TEXT
3  14
```

Summary

In this chapter, you learned about APL functions that sort data, manipulate data in a variety of ways, and perform math functions — from rounding numbers to solving simultaneous equations.

Unlike previous chapters, this chapter does not include specific exercises.

5
What Is a Program?

This chapter explains what a computer program is and why it is useful. It also provides a general understanding of how programs work and what they can do for you. You will get a chance to run some programs and to examine their parts. You will also be able to look at the data these programs use.

Before you begin this chapter, you must know how to install and start APL★PLUS on your computer. For information, read the Introduction to this book and the installation instructions provided with this system.

The examples in this chapter use programs and data located in the *DEMOAPL* workspace. If you want to perform the examples as you work through the chapter, be sure that this workspace is in your current working directory. The *DEMOAPL* workspace is generally located on a Utilities disk, in a workspaces directory, or on the original product disks or tape. If you need help locating the *DEMOAPL* workspace, refer to the system documentation or ask an experienced APL user to help you.

Programs and Data

The purpose of learning APL, or any other programming language, is to learn how to write programs. Once you learn to write programs, you can use a computer to do almost anything with numbers and words. A program is a list of instructions that tell the computer what to do. It is similar to using a recipe to bake a cake — the recipe contains a list of ingredients and steps that you must follow in a specific order. If you leave out an ingredient, miss a step, or perform the steps out of order, a cake is produced, but its taste and look are different from the cake you expected. Similarly, if the instructions in a program are incomplete or out of order, the computer will give you an answer, but it may not be the answer you expected or wanted.

You can store programs inside a computer or on a floppy disk, and use them over and over again. Once you write a program, it behaves the same way every time you use it.

For a program to *do* work, it must have information to work *on*. This information is called data. The ingredients you use to bake the cake (flour, eggs, and so on) correspond to the data that a program uses. Some programs use numbers as data (as desk calculators do); other programs use words and punctuation (as word processors do).

Loading a Workspace

In APL, programs and data are stored in workspaces. A workspace is similar to a folder in a desk drawer. To work with the documents in a file folder, you must first locate and retrieve the folder you want. Similarly, to use the data and programs in a workspace, you must first activate or load the workspace.

A ready-made workspace called *DEMOAPL* comes with the APL★PLUS System. Try loading this workspace. Type the command:

```
      )LOAD DEMOAPL
```

The system displays a message that includes the word *SAVED* and a date and time; for example:

```
SAVED 01/18/91 14:32:03
```

The system then displays the contents of the variable *DESCRIBE*. This variable describes the programs in the workspace. (**Note**: The text scrolls faster than you can read it. Press the Ctrl-S keys to stop the display; press the Ctrl-Q keys to continue the display.)

Note: If you use an APL system other than APL★PLUS or a computer other than an IBM, you may have to enter a slightly different version of the command. Refer to the system documentation for the correct command or ask an experienced APL user for help.

If you make a typing mistake when you enter the *)LOAD* command, the system displays error messages like *WS NOT FOUND* or *INCORRECT COMMAND*. If this happens, re-enter the command. Error messages are helpful because they tell you what went wrong.

Running a Program

After you load a workspace, you can use or run any of the programs in that workspace. You cannot run a program until the system displays the *SAVED* message.

Try running the *CALENDAR* program that is stored in the *DEMOAPL* workspace. The program prints a calendar for September (month number 9) of 1991.

Enter:

```
      CALENDAR 9 1991
```

The system displays the calendar for September 1991.

```
                  SEPTEMBER 1991

SUN     MON   TUES     WED   THUR     FRI     SAT

  1       2      3       4      5       6       7
  8       9     10      11     12      13      14
 15      16     17      18     19      20      21
 22      23     24      25     26      27      28
 29      30
```

Now, try running the program again with some other month, such as the month you were born.

To learn about other programs in the *DEMOAPL* workspace, enter:

```
          DESCRIBE
```

The system displays a description of the different programs in the workspace and how to use them. Run some of the programs to get a feel for the types of things programs can do.

The next section explains how to list and display programs.

Listing and Displaying Programs

You can display a list of the programs in the *DEMOAPL* workspace, or any other workspace, using the APL command *)FNS*. (*FNS* is an abbreviation for functions, which is what programs are often called in APL.) To list the names of all the programs, enter:

```
)FNS
```

The system displays the names of the programs. The length of the list depends on the workspace.

You can also see how a program *works*. That is, you can look at the parts of a program. You use the *LIST* command to display the contents of a program. For example, to list the contents of the program *CALENDAR*, enter:

```
LIST 'CALENDAR'
```

After you type *LIST 'CALENDAR'*, the system displays the *CALENDAR* program.

```
    ∇ Q←CALENDAR M;⎕IO;T
[1]   ⍝CALENDAR mm yy
[2]   ⍝prints specified month mm for year YY
[3]    ⎕IO←1 ⋄ T←M←M[⍒M],1 ⋄ M←1⌽3↑M
[4]    ⎕ERROR(21033≥ 0 12 ⊥M[3 1])/'NOT BEFORE OCT 1752'
[5]    Q←100|(M[1 3]← 1 0 +⌽ 0 12 ⊤¯3+ 0 12 ⊥M[3 1])[2]
[6]    Q←7|M[2]+Q+(⌊¯0.2+2.6×M[1])+(⌊Q÷4)-⌈1.75×⌊M[3]÷100
[7]    M←(T[2]∊ 4 6 9 11)+(T[2]=2)×3--/0= 4 100 400 4000 |1⍴T
[8]    Q←'BI6' ⎕FMT 6 7 ⍴42↑(Q⍴0),⍳31-M
[9]    Q←'    SUN   MON  TUES   WED  THUR   FRI   SAT',[1]Q
[10]   M←'JANUARY  FEBRUARY MARCH    APRIL    MAY      '
[11]   M←M,'JUNE     JULY     AUGUST   SEPTEMBEROCTOBER  '
[12]   M←(12 9 ⍴M,'NOVEMBER DECEMBER ')[T[2];]
[13]   M←(M⍳' ')↑M ⋄ M←M,⍕T[1] ⋄ M←((2+⌊(42-⍴M)÷2)⍴' '),M
[14]   Q←' ',[1](35↑M),[1]Q
    ∇
```

Note: If nothing appears on the screen, you probably made a typing error; try again. Be sure to include the two single quotes. Omitting a quote can cause your computer to freeze up. If this happens, enter the missing quote.

Look at the program. First, notice how *short* the program is. You may have seen programs written in other languages that were probably three or four times longer than this. APL is a very concise programming language — it allows you to do a lot in a very little space.

Next, notice the numbers in brackets to the left of the program. These numbers tell the computer the order of the steps. The computer must follow the steps of a program in the correct order to produce the correct result.

Finally, notice the symbols in the lines of the program. These are APL symbols, and each one does a particular job. The rest of this book explains what those symbols do and how you use them to make a program like *CALENDAR*.

You can display the contents of any program (function) just as you did with the program *CALENDAR*. To display a program in the *DEMOAPL* workspace, type the command *LIST* followed by the name of the function you want to see. Be sure to put single quotes around the function name; for example:

```
      LIST  'name'
```

Listing and Displaying Variables

You store data in variables — called that because the data inside them can vary. The programs in the *DEMOAPL* workspace use the data stored in variables to produce the effects mentioned in the *DESCRIBE* listing.

The contents of most of these variables may seem mysterious to you — a few odd numbers and words. It is not important now that you know how APL uses these numbers and words. You only need to know that the computer stores the data so that programs can use it.

You can display a list of the data in the *DEMOAPL* workspace by typing the command *)VARS*:

```
        )VARS
```

APL prints a list of names again, but this time it is a list of *data* names, separated by blanks.

To display the contents of a variable, you enter its name. For example, to display the contents of the variable named *MON*, enter:

```
        MON
```

You can display the contents of any variable in the list that is displayed when you use the command *)VARS*, but be sure to enter only one name at a time.

The remaining chapters in this book explain how to work with data in different ways and how to store your data.

clearing the workspace

When you load a workspace, it remains in the computer's memory until you either turn your computer off or remove the workspace. To remove a workspace from memory, you clear it or load a new workspac e. Before leaving this chapter, be sure to clear the *DEMOAPL* workspace.

To clear the workspace, enter the command:

```
      )CLEAR
```

The system displays the message

```
CLEAR WS
```

which tells you that you now have an empty workspace. If you enter

```
      )FNS
```

or

```
      )VARS
```

you see that the workspace is empty.

Summary

This chapter explained that:

- A program is a list of instructions that tells the computer how to perform work. A program is called a function in APL.

- A program uses data.

- You store data in a variable. You can see what is in a variable by entering its name.

- A workspace is used to store programs and data.

- You list the names of functions and variables using the commands *)FNS* and *)VARS*.

6

Writing Programs in APL

In this chapter, you will learn:

- how to enter function definition mode and create your own programs
- how to run your programs
- how to edit programs
- how to use both numeric and character data in programs
- about the six different classes of APL programs.

Note: The programs appearing in this chapter and subsequent chapters, including those in the exercises, are in the *LESSONS* workspace supplied with APL★PLUS systems. For information on how to copy the programs to your system, see the Introduction.

Creating a User-Defined Function

To create a user-defined function, you must first enter function definition mode, then create a function header and type in the lines that describe your function.

entering function definition mode

To enter function definition mode, type a del (∇), usually found on the G key. Follow this with the function name and the name of any data the function will use. Be sure to leave a space between the function name and the name of the data. To see how this works, enter:

```
            ∇AVG DATA
```

The system displays a number in brackets; for example:

```
[1]
```

entering lines of a function

The numbers in brackets indicate line numbers. In the example above, the [1] indicates that you have entered function definition mode and that the system is waiting for you to enter the first line of your function. Try it. Enter:

```
[1]     (+/DATA)÷ρDATA
```

After you press the Enter key, the system responds with the next line number; for example

```
[2]
```

which indicates that the system is now waiting for you to enter the second line of your function.

After you finish entering the lines that define your function, you leave function definition mode by typing another del (∇) symbol on the next numbered line. For example, when you leave the *AVG* function on line 2, your screen will look like this:

```
[2]     ∇
```

Some APL systems may require that you use a different sequence to leave function definition mode. Check your system documentation if this example does not work for you.

running a user-defined function

Test your function with some numbers you know the average of:

```
       AVG 90 100
95

       AVG 100 200 300
200
```

Now try:

```
       AVG 37 74 29
 46.66666667

       AVG 1.7 2.1 3.2
 2.333333333
```

Editing Functions

This section explains a few of the techniques you can use to edit a function. To edit a function, return to function definition mode by typing a del (∇) followed by the name of the function. For example:

```
          ∇AVG
```

replacing an entire line

The system displays [2] since it assumes that you want to add lines to the *end* of the existing function definition. To edit a different line, enter the number of the line you want to edit (in this case, line [1]):

```
[2]       [1]
```

The system responds with

```
[1]
```

which indicates that it is waiting for you to enter something *new* for line [1]. This new information replaces the old. Enter:

```
[1]       (+/DATA)÷⍴DATA
```

The system displays:

```
[2]
```

To leave function definition mode, enter:

```
[2]       ∇
```

You should now have a display similar to the following on your screen:

```
          ∇AVG
[2]       [1]
[1]       (+/DATA)÷⍴DATA
[2]       ∇
```

Test the function to make sure it works:

```
      AVG 25 50 75
50
```

replacing part of a line

You have learned how to edit a function by replacing an entire line with a new one. But you might want to change only part of the line, particularly if the expression is long. Create a new function to try this. Enter

```
      ∇ERASEBLANK S
```

which will delete all the blanks in a sentence. Now, enter the working part of the function, but make a mistake when you do it:

```
[1]   (' '≠S)/W
[2]   ∇
```

Notice that the *W* at the end of line [1] should be an *S*. If you try to run this function with the error in it, the system displays an error message:

```
      ERASEBLANK 'SO IT IS'
VALUE ERROR
      ERASEBLANK[1] (' '≠S)/W
                            ∧
```

APL displays the error message because you did not define a variable named *W*. Now that you know where the error is, you can re-enter function definition mode and correct it.

It is a good idea to enter the command)*SIC* before you edit a function that has a problem. The command)*SIC* clears (resets) the stack (a stack of things for the system to do), so that the system is not confused when you try to run the function after you have fixed the error. If you do not use the)*SIC* command, the system may display a definition error (*DEFN ERROR*) when you try to run the corrected function.

To correct the function, enter:

```
      ∇ERASEBLANK
```

The system displays:

```
[2]
```

You want the system to go to line [1], print it, and stop at the end of the line, you enter:

```
[2]    [1⎕0]
```

The only new character here is the quad (⎕) symbol, usually found on the L key. Quad symbols separate the desired line number from the desired position on the line (0, which means end of line). The system responds:

```
[1]    (' '≠S)/W
```

Press the Backspace key once. The system erases the letter W. Type an S and press the Enter key. When the system displays a [2], leave function definition mode by typing a del (∇):

```
[1]    (' '≠S)/S
[2]    ∇
```

Now, test the function:

```
      ERASEBLANK 'YOU CAN SAY THAT AGAIN'
YOUCANSAYTHATAGAIN
      ERASEBLANK '*  *  *  OOPS'
***OOPS
```

Displaying a Function Definition

Besides being able to edit the function, you will also want to display the function. First, you need to create a function. For this example, create a function that provides information about a bank account. The function should give you the largest deposit, the smallest deposit (which, if negative, is the largest withdrawal), and the present balance. Call the data *V* (for vector). Using what you have learned so far about creating functions, create the function *ACCOUNT* that is shown below.

```
        ∇ACCOUNT V
[1]     ⌈/V
[2]     ⌊/V
[3]     +/V
[4]     ∇
```

Suppose that the transactions on the account are deposits of $70, $130, and $35, withdrawals of $21 and $12, and another deposit of $27. To test the function, enter:

```
        ACCOUNT 70 130 35 ¯21 ¯12 27
```

Since the largest deposit is $130, the smallest deposit (the largest withdrawal) is -$21, and the present balance is $229, the system should display:

```
130
¯21
229
```

Now, suppose that the transactions on the account are a deposit of $30, a $22 withdrawal, deposits of $295 and $7, a $190 withdrawal, a $150 deposit, and a $55 withdrawal. Store these values in a variable called *A* and test the *ACCOUNT* function again:

```
        A←30 ¯22 295 7 ¯190 150 ¯55
        ACCOUNT A
```

Since in variable *A* the largest deposit is $295, the smallest deposit (the largest withdrawal) is -$190, and the present balance is $215, the system displays:

```
295
¯190
215
```

If you run this program several days from now, however, you might not remember what the three numbers in the result represent. To list the contents of the *ACCOUNT* function, return to function definition mode by typing a del (∇) followed by the name of the function:

```
      ∇ACCOUNT
```

The system displays `[4]` since it assumes that you want to add lines to the end of the existing function definition:

```
[4]
```

You do not want to add lines to the *ACCOUNT* function, but you do want to display it. Earlier you learned that you use the quad (⎕) symbol with numbers on both sides to go to a specific line number, print the line, and stop at the end of the line. You also use the quad (⎕) symbol to list the function. But since you want to list the entire function rather than one specific line, you do not need to surround the quad (⎕) symbol with numbers. You enter only `[⎕]`. Try it. Enter:

```
[4]   [⎕]
```

The system lists the function, including the first line (the name line), and again waits for you enter line `[4]`:

```
[0]   ACCOUNT V
[1]   ⌈/V
[2]   ⌊/V
[3]   +/V
[4]
```

To leave function definition mode, enter a del (∇) symbol:

```
[4]   ∇
```

Inserting a Line

Besides being able to edit and display a function, you will also want to add lines to a function. Suppose you want to calculate the running balance in the function *ACCOUNT*, and that you want this new line to appear immediately after line [2]. To insert the line, first enter function definition mode:

```
          ∇ACCOUNT
```

The system again displays [4] since it assumes that you want to add lines to the end of the existing function definition:

```
[4]
```

But you want to insert the new line between the existing lines 2 and 3. To insert the line between line [2] and [3], enter any number between 2 and 3; for example, [2.5]:

```
[4]    [2.5]
```

The system displays the new line number

```
[2.5]
```

and waits for you to enter the text for line [2.5]. On the new line, you enter the expression for calculating a running balance:

```
[2.5]    +\V
```

You have nothing else to add to the *ACCOUNT* function, but you want to verify that the system inserted the line. To display the function, enter [⎕]:

```
[2.6]    [⎕]
```

The system inserts the new line between lines [2] and [3] and gives it the line number you entered. Then it displays the function:

```
[0]     ACCOUNT V
[1]     ⌈/V
[2]     ⌊/V
[2.5]  \V
[3]     +/V
[4]
```

To leave function definition mode, enter a del (∇):

```
[4]     ∇
```

Using the transactions that you stored earlier in variable *A*, test the changes you made to the function. Enter:

```
        ACCOUNT A
```

The display still shows the largest and smallest deposits and the present balance. But now it also includes another item, the running balance:

```
295
¯190
30 8 303 310 120 270 215
215
```

Display the *ACCOUNT* function again. Enter:

```
        ∇ACCOUNT
[5]     [⎕]
```

The system displays:

```
[0]     ACCOUNT V
[1]     ⌈/V
[2]     ⌊/V
[3]     +\V
[4]     +/V
[5]
```

Notice that line [2.5] is now line [3]. The system automatically renumbers the function lines when you leave function definition mode. This makes it easier to keep track of line numbers when you begin writing and editing longer functions.

Erasing a Line

Being able to add and edit function lines gives you a lot of flexibility when you are creating your own functions. However, there are times when you want to delete an entire line. Take line `[4]` in the *ACCOUNT* function, for example. Since the running balance in line `[3]` also shows the ending balance, you do not need to include line `[4]` in the function. To delete a line, you use a tilde (~) before the line number. (The tilde (~) is usually found on the T key.) To delete line `[4]` in the *ACCOUNT* function, you enter:

```
[5]    [~4]
```

The system displays:

```
[4]
```

List the function again to make sure that the system deleted line `[4]`. Enter:

```
[4]    [⎕]
```

The system displays:

```
[0]    ACCOUNT V
[1]    ⌈/V
[2]    ⌊/V
[3]    +\V
[4]
```

Now leave function definition mode by entering a del (∇):

```
[4]    ∇
```

Summary of Editing Commands

Table 6-1 summarizes the five editing commands presented in this chapter.

Table 6-1. Function Editing Commands

Command	Description
`[n]`	Sends the system to line `[n]`. The expression you enter replaces the contents of the existing line `[n]`.
`[n⎕0]`	Sends the system to line `[n]`, prints the line, then waits at the end of the line for input. Use the Backspace key to edit the line.
`[⎕]`	Displays the entire function.
`[n.m]`	Inserts a line between two existing lines.
`[~n]`	Deletes line `[n]`.

Adding Character Data to Functions

When you began this chapter, you created a function called *AVG*. What did the *AVG* function do? You can display the *AVG* function to remind you what the result represents. But it would be more useful if *AVG* displayed a message explaining what the results represent. For example

```
      AVG 2 3 4
THE AVERAGE IS 3
```

is more informative than

```
      AVG 2 3 4
3
```

and easier than displaying and interpreting the *AVG* function:

```
      ∇AVG
[2]   [1]
[1]   (+/DATA)÷ρDATA
[2]   ∇
```

How do you create this kind of result? In an earlier chapter of this book, you learned how to use the format (⍕) and catenate (,) functions to combine characters and numbers in immediate execution mode. Before you try adding character data to functions, briefly review some of the guidelines for combining characters and numbers:

- To display character data in APL, you surround characters with single quotes.

- To display numbers with character data, you use the format function (⍕) to convert the numbers to characters and then use the catenate function (,) to combine the converted data with the character data.

- You can put an APL expression on the right side of the format symbol. APL performs the arithmetic first, converts the answer to character data, then catenates it to the sentence.

Now, using these guidelines, create the *ACCOUNT2* function shown below. This function is similar to *ACCOUNT*; however, the new function contains character data that identifies what each of the results represent. Enter:

```
      ∇ACCOUNT2 DATA
[1]   'LARGEST DEPOSIT IS ',⍕⌈/DATA
[2]   'LARGEST WITHDRAWAL IS ',⍕⌊/DATA
[3]   'RUNNING BALANCE IS ',⍕+\DATA
[4]   ∇
```

You may want to list your program to make sure you entered everything correctly. In particular, make sure there is a quote at the beginning and end of each character string.

Now test the *ACCOUNT2* function with deposits of $78 and $122, withdrawals of $23, $11, and $55, and a $67 deposit. Enter:

```
      ACCOUNT2 78 122 ¯23 ¯11 ¯55 67
```

The system should display:

```
LARGEST DEPOSIT IS 122
LARGEST WITHDRAWAL IS ¯55
RUNNING BALANCE IS 78 200 177 166 111 178
```

If the system displays any error messages, enter the command)*SIC* and then enter function definition mode. Return to the line where the system showed the error and correct it. You may find that once you correct an error in line [1], the system will find another error in lines [2] or [3].

Different Function Forms

The functions you have created up to this point have the same basic form:

function name data

For example, the *AVG* and *ERASEBLANK* functions you created in this chapter use only one argument — data to the right of the function name:

```
      AVG DATA
      ERASEBLANK S
```

Many functions, however, use two arguments; that is, they use data on both sides. You already know that the divide (÷) and compress functions (/) use data on both sides:

```
      6÷2
3

      0 1/'AZ'
Z
```

Now imagine that you want to calculate the perimeter of a rectangle. To calculate the perimeter, you add the lengths of two adjacent sides and multiply by 2. Therefore, a function called *PERIMETER* would need two numbers (arguments) to calculate the perimeter of a 5-by-6 rectangle — the width (5) and the length (6); that is:

```
      5 PERIMETER 6
```

It is as easy to create functions with two arguments as it is to create functions with one argument. Try it. Create the *PERIMETER* function. First, enter:

```
      ∇WIDTH PERIMETER LENGTH
```

When you enter three names separated by blanks, you tell the system that:

- the name of the function is *PERIMETER*
- one piece of data (the left argument) is called *WIDTH*
- a second piece of data (the right argument) is called *LENGTH*.

Now enter the *PERIMETER* function as follows:

```
[1]    2×WIDTH+LENGTH
[2]    ∇
```

To test the *PERIMETER* function, enter:

```
       10 PERIMETER 20
```

The system should display:

```
60
```

Now enter:

```
       5 PERIMETER 8
```

The system should display:

```
26
```

APL functions can use either one or two pieces of data. Try to create the *VOLUME* function shown below.

```
       ∇WIDTH VOLUME LENGTH HEIGHT
```

You get a *DEFN ERROR* message and return to immediate execution mode. Why? When you enter two names (such as *AVG V* or *ERASEBLANK S*), the system assumes that the function name is the name on the left. When you enter three names (such as *WIDTH PERIMETER LENGTH*), the system assumes that the name in the middle is the function name, and that the two names on either side are data names. But when you enter more than three names, APL does not know which name is which.

If the system allows a maximum of two arguments, how can you perform calculations that require *more* than two arguments? For example, how can you calculate the volume of a cube if you need three numbers for the calculations — the length, width, and height? You define a function called *VOLUME* that has a single argument

```
      VOLUME LWH
```

where *LWH* is a three-element vector containing the length, width, and height.

Functions with No Arguments

You know now that you can create a function that has one argument or two arguments. But there is another type of function you can create — a function with *no* arguments.

Create the *DIE* function shown below. This function uses the built-in APL function called roll (?) to produce a random number between 1 and 6 (much like rolling a die).

```
      ∇DIE
[1]   ?6
[2]   ∇
```

To test the *DIE* function, enter only the function name. The *DIE* function does not require any data.

```
      DIE
```

What does the system display? Since the *DIE* function produces a random number between 1 and 6, the system can display any number in that range. For example, your display may look like

```
      DIE
4
```

or:

```
      DIE
1
```

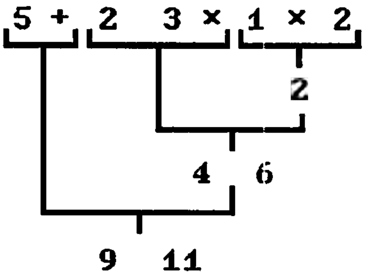
Functions that take no data are useful for tasks like printing a report or controlling files. You will learn more about functions with no arguments in later chapters.

Explicit Results

In the first four chapters, you learned how to use the functions that the system provides — functions like plus (+), less than (<), maximum (⌈), compression (,), and membership (∊). Since these functions come with APL★PLUS, they are called built-in functions. One important characteristic of the built-in APL functions is that they return explicit results. This means that one APL function provides numbers that other APL functions can use. For example, you can visualize the statement

```
5+2  3×1×2
```

as:



As you can see, the system calculates two explicit results before it calculates the final result:

```
9  11
```

You never see the explicit results (such as 4 6) on the screen, because the next portion of the statement uses it immediately.

User-defined functions behave and perform the same as the functions that are built into your APL system. In other words, you should be able to enter 2+*AVG* 9 10 11 and get an answer. But what happens if you try it?

```
      2+AVG 9 10 11
10
VALUE ERROR
      2+AVG 9 10 11
        ^
```

The problem is that you have not defined *AVG* to return an explicit result. The *AVG* function displays the answer to the *AVG* 9 10 11 portion of the expression (10) and then promptly "forgets" it since the answer is not stored anywhere. Without an explicit result, the system cannot add the 2.

Defining Functions with Explicit Results

The problem described above is easy to fix. You need to tell the function to store the result in a variable. Try it. Define a new version of *AVG* that will store the result of the function in a variable *R* (for "result"). Enter:

```
      ∇R←AVG2 A
[1]
```

Enter the rest of the function, then leave function definition mode:

```
[1]   R←(+/A)÷ρA
[2]   ∇
```

Again, notice that the APL expression on line [1] assigns the result to the variable *R*. If you want the function to have an explicit result, you must specify the name of the result variable in the header (name line) of the function and assign the result data to the result variable in the body of the function. The name of the result variable must be the same in both places.

Test the new function. Enter:

```
      2+AVG2 9 10 11
```

The system displays:

```
12
```

Now try using the results of one function in another function. Calculate the perimeter of a rectangle whose width is 2 and whose length is the average of the values 9, 10, and 11. Using the *AVG2* and *PERIMETER* functions, enter:

```
      2 PERIMETER AVG2 9 10 11
```

AVG2 returns the result 10, then 2 *PERIMETER* 10 gives the answer:

```
24
```

One function (*AVG2*) gives a result, which is then passed on to the next function to give the "final" answer. The system displays only the final answer.

Since the new *AVG2* function returns an explicit result, you can save the result in a variable:

```
      ANS←AVG2 9 10 11
```

To display the answer, enter:

```
      ANS
```

The system displays:

```
10
```

This was impossible to do with the original function *AVG*, for the same reasons explained above:

```
      ANS←AVG 9 10 11
10
VALUE ERROR
      ANS←AVG 9 10 11
         ^
```

Try it again. Create alternative versions of two other functions:

```
      ∇R←WIDTH PERIM LENGTH
[1]   R←2×WIDTH+LENGTH
[2]   ∇
```

```
        ∇ANS←DIE2
[1]     ANS←?6
[2]     ∇
```

Now you can use these functions as arguments to each other. For instance, you can "roll two dice" and show the result:

```
        DIE2+DIE2
6

        DIE2+DIE2
9
```

You can also assign their results to variables.

```
        Q←4 PERIM 6
        Q
20
```

Function Syntax

Table 6-2 summarizes the six different forms of functions you learned in this chapter. These are all forms of functions that are defined in APL. Another word for the form of a function is syntax. Table 6-2 shows the syntax of each function type (using *FN* as the function name, *R* as the result, and *A* and *B* as argument names) and also shows examples of each type of function you created in this chapter.

Table 6-2. Types of Function Syntax

No Explicit Result	Explicit Result
No Arguments	
FN *DIE*	*R←FN* *R←DIE2*
One Argument	
FN A *AVG DATA*	*R←FN A* *R←AVG2 A*
Two Arguments	
A FN B *WIDTH PERIMETER LENGTH*	*R←A FN B* *R←WIDTH PERIM LENGTH*

Before you go on to the exercises, store your functions and variables with the command:

```
)SAVE MYWORK
```

Summary

In this chapter, you learned that:

- You should create your own functions, because you can:
  - ❒ save typing
  - ❒ perform more complicated tasks
  - ❒ store them for future work.
- You enter function definition mode by entering a del (∇) and leave it by entering another del.
- A function consists of a "header," which names the function and identifies the data, and a "body," which does the work.
- You edit functions using the commands listed in Table 6-1.
- You can use both numeric and character data in a function — just like in immediate execution mode.
- Functions can have no arguments, one argument, or two arguments.
- To use the results of a function in other calculations, you must define the function to return an explicit result.

Exercises

A. 1. Create the following function. This is just a test program you can use to practice your editing skills. If you run it, APL displays a *VALUE ERROR* message.

```
       ∇NUMBERS
[1]     ONE
[2]     TWO
[3]     THREE
[4]     FOUR
[5]     FIVE
[6]     ∇
```

2. Enter function definition mode and replace line [2] of *NUMBERS* with the word *TOO*.

3. Replace line [4] with the word *FORE*.

4. Using the editing command [*n*⎕0], change line [3] to read *TREE*.

5. Add a line between lines [1] and [2] containing the word *TWO*.

6. Delete line [5].

7. List the function.

8. Change the name of the function to *COUNT*. (Use the command [0] to do this — line [0] is the function header.)

9. List the function.

10. Leave the function.

B. 1. On a piece of paper, write the header for a function called *ADD*. *ADD* should have two arguments and return an explicit result. The function should work as follows:

```
      4 ADD 5
9
        ANSWER←4 ADD 2
        ANSWER
6
```

2. Write the body of the function *ADD*.

3. Create the function *ADD* on the system and test it.

4. Write (on a piece of paper) the header for a function *RUNTOT* that takes one argument and returns a running total of the data in the argument. The function should not return an explicit result.

5. Write the body of the function *RUNTOT*.

6. Create the function *RUNTOT* on the system and test it. Use the numbers 10, 20, 30, 40.

C. 1. Create a function called *HI*, which works as follows:

```
      HI
HELLO.  MY NAME'S APL.
WHAT'S YOURS?
```

The function does no arithmetic; it only prints the message.

2. Create a function called *HALF* that returns one-half of the number (or numbers) you give it. Use the function to find one-half of 3.14159.

3. Redefine *HALF* so that it returns an explicit result.

4. Use the function *HALF* as part of an APL expression to find the sum of 10 and one-half of 3.14159.

5. Define a function called *ROWTOT* that returns the row totals for a matrix; for example:

```
      MAT←3 2⍴⍳6
      MAT
1 2
3 4
5 6

      ROWTOT MAT
3 7 11
```

6. Define a function called *COLTOT* that returns column totals; for example:

```
      COLTOT MAT
9 12
```

7. Try the *ROWTOT* and *COLTOT* functions on the following matrix:

```
      MAT2←3 4⍴10+⍳12
      MAT2
 11 12 13 14
 15 16 17 18
 19 20 21 22
```

8. Use the *COLTOT* function in a new function *RPT*, which displays a simple report based on a matrix. The report should:

   - print the data (the matrix)
   - print a line of underlines (hyphens)
   - print the column totals (use *COLTOT MAT2*).

   The function should produce the following display:

```
      RPT MAT2
 11 12 13 14
 15 16 17 18
 19 20 21 22
-----------
45 48 51 54
```

   (Notice that the columns do not line up. You will learn how to align columns later in this book.)

D. 1. Create a function *REPEAT* that produces the following display:

```
          REPEAT 'ANN'
ANN ANN ANN ANN ANN

          REPEAT 'JO'
JO JO JO JO JO
```

(Hint: Use the catenation function (,).)

2. You can find the square of a number by multiplying it by itself:

```
          3×3
9
```

Create a function called *SQUARE* that produces the following display:

```
          SQUARE 9
THE NUMBER SQUARED IS 81
```

What is the square of 247?

3. Create a function *CHOOSE* that has the following effect:

```
          8 7 12 73 55 CHOOSE 3
12

          101 25 14 47 CHOOSE 2
25
```

The function should choose a number from the left argument based on the position specified by the right argument. The function should also work if you use it as follows:

```
          NOS←8 76 43 21 23 6
           NOS CHOOSE 4
21
```

(Hint: Use the indexing function ([]).)

4. Create a function called *OF*. *OF* has two arguments and works as follows:

```
      20 OF '*'
********************

      4 OF '⎕'
⎕⎕⎕⎕
```

OF displays as many of the items on the right as you ask for with the number on the left. You should also be able to enter:

```
      2 3 OF '?'
???
???
```

(This should give you a clue as to how the function works.)

5. A histogram is a type of bar chart. Use the *OF* function in a new function, *HIST*, to produce a horizontal histogram, as shown below:

```
      HIST 4 1 6 7
⎕⎕⎕⎕
⎕
⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕
```

(Hints: Use *OF* with a right argument of '⎕' on four different lines of the function to produce the display. Use indexing to pick numbers out of the right argument to *HIST* one at a time. The function should work for four numbers only; that is, not for three, five, and so on.)

E. 1. Create a function *GT* that produces the following results:

```
          SALES←125 350 201 115 279
          SALES GT 200
350 201 279
```

GT chooses numbers from the left argument that are greater than the number in the right argument. Make sure that your function returns an explicit result. Store the result in the variable, *BIGSALES*, as follows:

```
          BIGSALES←SALES GT 200
          BIGSALES
350 201 279
```

2. Create an *LT* function that is similar to *GT*, except that it selects numbers less than the right argument. Then try the expression *SALES LT* 200.

3. Create the function, *EQ*, that selects numbers equal to the right argument. Then try the expression:

```
          SALES EQ 350.
```

F. Try to create the following system. It should consist of four functions:

```
INITBAL
CHECK
DEPOSIT
NEWBAL
```

This system will:

- set an initial bank balance
- keep a record of checks
- keep a record of deposits
- show the new balance.

All of the functions use the same two variables, *DEPOSIT* (deposits) and *WITHD* (withdrawals).

The first function is shown below.

```
      ∇INITBAL AMOUNT
[1]   DEPOSIT←AMOUNT
[2]   WITHD←0
      ∇
```

Use this function once when you start the program. (Or, you could use it at the end of each month to "clear out" your account and set the balance for the next month.) *INITBAL* sets *DEPOSIT* to the amount you specify, and it sets *WITHD* to 0.

The following example demonstrates how these functions work together.

```
      INITBAL 100
      DEPOSIT 25 50
      NEWBAL
175
      CHECK 20 40 10
      NEWBAL
105
```

Hints:

1. *CHECK* and *DEPOSIT* should have the forms:

   ```
   ∇CHECK AMOUNT
   ∇DEPOSIT AMOUNT
   ```

2. Remember that you can add data to the end of a vector using the catenate function; for example:

   ```
   DATA←DATA,25
   ```

   This expression catenates the number 25 to the end of an existing string of numbers.

3. *NEWBAL* should have the form:

```
∇NEWBAL
```

This function takes no arguments. It merely looks at *DEPOSIT* and *WITHD* and reports the present balance. (Use +/ in this function.)

7
APL Programming Techniques

In this chapter, you will learn:

- how to create interactive programs — functions that display messages requesting the information they need to complete a task
- how to use the branch function to create loops
- how to use APL programming aids such as line labels and conditional branches.

Creating Interactive Functions

Up to this point, you have supplied data when you used a function. In immediate execution mode, you entered statements like `10-4` and `8⌊10`. After creating a function in function definition mode, you entered statements like *AVG* `9 10 11` and `5` *PERIMETER* `6`.

But it is just as easy to have a function ask you for the data it needs. Functions that ask for data are called interactive functions because they interact with you.

Evaluated Input

Suppose that you want to create a function that averages numbers that you provide when asked. You only need to include the statement *variable*`←⎕` in the function, where *variable* is any variable name you choose.

Create the function *ASKAVERAGE*:

```
      ∇ASKAVERAGE
[1]    'WHAT NUMBERS DO YOU WANT TO AVERAGE?'
[2]    VECTOR←⎕
[3]    ANSWER←(+/VECTOR)÷⍴VECTOR
[4]    'THE AVERAGE IS ',(⍕ANSWER),'.'
[5]    ∇
```

Notice that the function does not have any arguments. You enter the data when the function asks for it. *ASKAVERAGE* stores the data in the variable *VECTOR*, then it uses the data in *VECTOR* to calculate the average.

Try using the *ASKAVERAGE* function. Enter:

```
      ∇ASKAVERAGE
```

The system displays:

```
WHAT NUMBERS DO YOU WANT TO AVERAGE?
⎕:
```

quad input mode

Notice the quad-colon symbol (⎕ :). When the system displays this symbol, it is in evaluated input mode (also called quad input mode). The function uses the quad-colon symbol (⎕ :) to ask you to enter numbers or a variable containing numbers. In response to the quad-colon prompt, enter:

```
      8 10 12
```

The *ASKAVERAGE* function stores the data in the variable *VECTOR*, then uses the data in *VECTOR* to calculate the average:

```
THE AVERAGE IS 10.
```

Character Input

You can enter character data in quad input mode; however, you must enclose the data in single quotes. Suppose the you want to create a function that welcomes you by the name you provide when asked. You only need to include the statement *variable*←⎕ in the function, where *variable* is any variable name you choose. Create the function *HELLO*:

```
      ∇HELLO
[1]   'MY NAME IS JERRY.  WHAT''S YOURS?'
[2]   A←⎕
[3]   'NICE TO MEET YOU, ',A,'.'
[4]   ∇
```

Like the *ASKAVERAGE* function, the *HELLO* function does not have any arguments, and you enter the data when the function asks for it. *HELLO* stores the data in the variable *A*, then uses the data in *A* to welcome you by name.

Try using the *HELLO* function. Enter:

```
      HELLO
```

The system displays:

```
MY NAME IS JERRY. WHAT'S YOURS?
```

(blank line)

quote-quad
input mode

Unlike quad input mode, the system does not display a special symbol. The system moves to the next line (without a six-space indent) and waits for you to enter data. When the system displays the blank line, it is in character input mode (also called quote-quad input mode because of the quote-quad symbol () that you use in the function). Quote-quad input mode accepts letters and other characters without quotes.

On the blank line, enter:

```
JODY
```

The system stores the data in the variable *A* and displays:

```
NICE TO MEET YOU, JODY.
```

Using the quad and quote-quad input techniques, you can make functions much easier to use.

testing for an
empty result

Suppose that rather than typing *JODY*, you had pressed the Enter key. What would happen? The system would display:

```
NICE TO MEET YOU, .
```

There is an APL idiom, useful when you use ⍞ input, that checks whether the user pressed the Enter key instead of typing something. The idiom for checking for an empty vector is

```
        0=⍴A
```

You could include this idiom in the *HELLO* function to check for an empty vector, then leave the program rather than display the greeting. (This requires branching techniques that are discussed later in this chapter.)

accepting numeric quote-quad input

Suppose that you want to develop a function that averages values collected in quote-quad input mode. Since quote-quad input mode allows both characters and numbers, how can you ensure that the data contains only numbers? You need to use two system functions that you learned about in Chapter 4: ⎕*FI* (format inverse) and ⎕*VI* (verify input). ⎕*FI*:defined converts valid numbers from character data to numeric data, and changes invalid numbers to 0. ⎕*VI* tells you what a character string contains; the system displays 1s for valid numbers and 0s for everything else.. When you use ⎕*VI* and ⎕*FI* together, you can collect numeric input using ⍞ rather than ⎕. You use ⎕*VI* to verify that the entry contains valid numbers. If it does, you then use ⎕*FI* to convert the entry to numbers.

For example, to ignore character data and extract valid numeric data from a variable *DATA*, you can use the statement:

```
      (⎕VI DATA)/⎕FI DATA
```

The ⎕*VI* expression in the left argument generates 1s for valid numbers and 0s for invalid numbers. Then, the ⎕*FI* expression in the right argument converts the character data to numeric data, replacing invalid numbers with 0s. The compression function uses the 1s and 0s as a selection expression on the right argument.

Try the following example:

```
      DATA←'12.1 HI 52 ⍉⍋'
      (⎕VI DATA)/⎕FI DATA
12.1 52
```

How does the statement work? Visually, the calculation looks like:

```
      1 0 1 0/12.1 0 52 0
12.1 52
```

Escaping from Quad and Quote-Quad Input

In most cases, you can interrupt a function to make a function stop (if you want to get out of it for any reason). However, this feature does not work with evaluated input mode and character input mode. The system does not allow you to interrupt input; it prints ⎕:, or jumps to the next line, until you enter something.

To leave evaluated input mode, enter a right arrow (→). The techniques for leaving character input mode vary among APL systems. Refer to your system documentation for more information. When you leave either mode, the system returns you to immediate execution mode.

The next section introduces another built-in APL function that can help you use quad and quote-quad input.

Looping

Looping is a way to make the system do the same thing over and over again. Suppose, for example, that you want the system to simulate flips of a coin. You can create a function that flips a coin and, after displaying the result, asks if you want to try again. If you respond yes, the system repeats the process.

branching

You use the branch function (→) to create a loop in APL. This function sends the system to another line in the program. For example, to send the system to line [2] of a function, include the following expression in the function:

```
→2
```

Each time the system reads the branch function, it goes to line [2] and performs whatever APL functions are on that line and the lines that follow. In the coin flipping function, however, you only want to repeat the process if a certain condition is met: if the response to the try again prompt is yes.

conditional branch

You can create an expression that is a test or condition to the branch function. The system performs the branch depending upon the outcome of the test. To create a conditional branch, include a test with the branch function. For example, in the coin flipping function, you can include the conditional branch:

```
→('Y'=1↑⍞)⍴2
```

The test appears between the two parentheses so that the system executes it first. The test checks to see whether the first letter of quote-quad input is a Y. If it is, the answer is true, or 1:

```
→1⍴2
```

The expression 1ρ2 results in

```
→2
```

and the system branches to line [2]. If you enter anything other than a *Y* (*NO*, for instance), the answer to the test is false, a 0:

```
→0ρ2
```

Reshaping data with 0 creates a vector with no elements. The system does not perform the branch and the function continues on to the next line.

Now create the function *FLIP* to see how looping works. Enter:

```
        ∇ FLIP
[1]     N←2 5ρ'HEADSTAILS'
[2]     ((?2)=1 2)/N
[3]     'TRY AGAIN?  (YES OR NO):'
[4]     →('Y'=1↑⍞)ρ2
[5]     ∇
```

FLIP "flips a coin" and then displays the result of the flip, either *HEADS* or *TAILS*. *FLIP* pauses and asks you if you want to run the function again or exit. Try running the function.

```
        FLIP
HEADS
TRY AGAIN?  (YES OR NO):
YES
TAILS
TRY AGAIN?  (YES OR NO):
NO
```

When you answer yes, *FLIP* goes back to line [2] of the function and runs again; that is, it loops.

conditional branch to next line

You aready know that the idiom for conditionally branching to another line in the program is:

```
→(CONDITION)ρLINE1
```

But how do you conditionally branch to the next line in a function? An idiom for "go to the next line if the condition is met" is:

```
→(CONDITION)ρ⎕LC+1
```

The system function ⎕*LC* is the line counter, and it gives you the line number of the function that is currently being executed. So if you go to ⎕*LC*+1, you are going to the next line. The following example shows this technique.

```
  .
  .
  .
[6]  →(NUM>0)ρ⎕LC+1 ◊ 'NONE FOUND.' ◊ →EXIT
[7]  (⍕NUM), ' OCCURRENCES FOUND.'
  .
  .
  .
[20]  EXIT:
[21]  ∇
```

If *NUM* is not larger than 0, the system finds nothing, so APL prints a message and leaves the program. Otherwise, the system goes to line 7 (⎕*LC*+1) and reports on the number of occurrences found. (Note that *EXIT* is an example of a line label. Line labels are discussed in the next section.)

Line Labels

Each line in an APL function has a unique line number appearing on the far left side. In the *FLIP* function, you used a line number to tell the system which line to branch to. You can also give names, called labels, to function lines. You should use line labels to do branching. The technique is simple — you label certain lines of a function and use the labels in place of line numbers in branch statements.

The form for line labels is:

label: *APL expression*

Use a colon to separate the line label from the APL statement on the same line. You can use any name for the label, but you cannot have a label with the same name as a variable; be sure to pick a unique name. It is best to use names that reflect what the function is doing — like *END*, *LOOP*, and so on.

The next example shows what the *FLIP* function looks like with line labels.

```
      ∇FLIP
[1]    N←2 5ρ'HEADSTAILS'
[2]    MORE: ((?2)=1 2)/[1]N
[3]    'TRY AGAIN? (YES OR NO):'
[4]    →('Y'=1↑⍞))ρMORE
      ∇
```

Notice that the label *MORE* appears on line [2], and that on line [4] the label *MORE* replaces the line number. Instead of branching to line [2], *FLIP* now branches to the line labeled *MORE*.

Line labels make a function much easier to create and edit. For example, if you decide to add a line to the beginning of *FLIP*, the current line [1] becomes line [2], line [2] becomes line [3], and so on. If you use line numbers in your branch statement (line [4]), the function branches to the wrong line. To correct this, you must edit the branch statement. But if you use line labels, the branch always goes to the correct line, since it looks for the *name* instead of the line number.

As another example, create the following function *GUESS*:

```
        ∇GUESS
[1]     'I PICKED A NUMBER BETWEEN 1 AND 3.'
[2]     'WHAT IS IT?'
[3]     LOOP: 'ENTER YOUR NUMBER:'
[4]     NUM←⎕
[5]     ANS←?3
[6]     →(ANS=NUM)ρOK
[7]     'SORRY.  THE NUMBER WAS ',(⍕ANS),'.'
[8]     'TRY AGAIN?'
[9]     →('Y'=1↑⍞))ρLOOP
[10]    →0
[11]    OK: 'RIGHT!  TRY AGAIN?'
[12]    →('Y'=1↑⍞))ρLOOP
[13]    ∇
```

Now, try to run the *GUESS* function. Your display will be similar to the next example.

```
      GUESS
I PICKED A NUMBER BETWEEN 1 AND 3.
WHAT IS IT?
ENTER YOUR NUMBER:
⎕:
      3
SORRY.  THE NUMBER WAS 2.
TRY AGAIN?
YES
I PICKED A NUMBER BETWEEN 1 AND 3.
WHAT IS IT?
ENTER YOUR NUMBER:
⎕:
      3
RIGHT!  TRY AGAIN?
YES
I PICKED A NUMBER BETWEEN 1 AND 3.
WHAT IS IT?
ENTER YOUR NUMBER:
⎕:
      3
SORRY.  THE NUMBER WAS 1.
TRY AGAIN?
NO
```

The *GUESS* function includes two tests — one to check the number you enter, and one to check for a *YES* or *NO* answer. Line [5] calculates the system's answer by generating a random number between 1 and 3. Line [6] checks to see if the system's answer and your guess are the same. If so, the system goes to line [11], displays *RIGHT!*, and asks if you want to try again. If you enter *YES*, the system returns to line [3]. If you enter *NO*, it stops.

If the test in line [6] shows that the answer and your guess are not equal, the system continues to line [7], displays the correct answer, and asks if you want to try again. Line [9] uses the same test as line [12]. If your answer is *NO*, the system goes to line [10]. This line says "go to 0," which tells the system to leave the function.

Note: Now that you know how to use loops, you may accidentally create an endless loop that the system cannot stop. An endless loop occurs when you forget to include a test , or exit, from the loop. If you notice that the system is taking too long to run a function, you can interrupt the function (usually with Ctrl-Break). The system prints the line number where the function stopped. You can then list the function and see if something is wrong. Remember to enter

```
      )SIC
```

before you edit the function; otherwise, the system displays a *DEFN ERROR*.

Programming Aids

This section introduces more programming tools to make your functions easier to use and understand. These tools are comments and local variables.

Comments

In a function, a comment is a statement that the system does not execute. You can use comments to describe the purpose of a function, the steps it performs, or a complicated expression.

lamp

To tell the system that what you are entering is a comment, enter a lamp symbol (⍝) before the comment text. This symbol (usually formed with Alt-,) is called a lamp because it precedes the text that illuminates what is happening in the function. For example, line [1] of the following function includes a comment.

```
       ∇NOBLANK CHARVECT
[1]    ⍝ REMOVE BLANKS FROM CHARACTER VECTOR
[2]    (CHARVECT≠' ')/CHARVECT
[3]    ∇
```

Local and Global Variables

You frequently use variables inside functions to store data. You also store data in variables in the workspace. If you use the same name for a variable in a function as you do for a variable in a workspace, you run the risk of destroying the data in the workspace variable. To see how, create the following function *COMPUTESALES*. This function computes sales data for a week given the units sold.

```
      ∇COMPUTESALES
[1]   ⍝ COLLECT UNITS SOLD, COMPUTE SALES
[2]   'ENTER SALES FOR THIS WEEK:'
[3]   SALES←⎕
[4]   'TOTAL SALES ',⍕+/SALES×PRICE
      ∇
```

Now, create the *PRICE* variable and the *SALES* variable:

```
      PRICE←10.30
      SALES←105
```

Run the function:

```
      COMPUTESALES
ENTER SALES FIGURES FOR THIS WEEK:
⎕:
```

Enter some sales figures; for example:

```
      175.24 159.4 211.35 201.56 213.45
```

The system displays:

```
TOTAL SALES 9898.3
```

Line `[3]` of *COMPUTESALES* assigns your response to *SALES* and replaces whatever was in *SALES* before. Display *SALES* again:

```
      SALES
175.24 159.4 211.35 201.56 213.45
```

The function replaced your old variable *SALES* with the new figures.

To prevent a function from replacing data, you can use local variables. Local variables allow you to lock variables inside a function. Local variables are meaningful only in the context of a particular function. In contrast to local variables, workspace variables are known as global variables.

To create a local variable, add the variable name to the function header. Separate the function name and variable name with a semicolon; for example:

```
      ∇COMPUTESALES
[6]   [0⎕0]
[0]   COMPUTESALES;SALES
[6]   ∇
```

Now, redefine *SALES* in the workspace:

```
      SALES←105
```

Run the function again:

```
      COMPUTESALES
ENTER SALES FIGURES FOR THIS WEEK:
⎕:
```

Enter some different sales figures:

```
      10 20 30 40 50
```

The function assigns the sales figures to the variable *SALES*, but this time *SALES* remains separate, stored in the function. The system displays:

```
TOTAL SALES 1545
```

Display the workspace variable *SALES*:

```
      SALES
105
```

The function did not change the number in the workspace variable *SALES*.

Using local variables lets you protect variables you create in the workspace. The technique also allows you to use the same variable names in many functions. For example, a sales tracking system might contain many variables called *SALES* — one global variable and many local variables locked inside functions.

Local variables also save room in the workspace, because the system erases local variables as soon as the function is finished running. To see how this works, erase *SALES*:

```
      )ERASE SALES
```

When you try to display *SALES* now, you get a *VALUE ERROR*.

```
      SALES
VALUE ERROR
      SALES
      ^
```

Now run *COMPUTESALES*.

```
      COMPUTESALES
ENTER SALES FIGURES FOR THIS WEEK:
⎕:
```

Enter some sales figures:

```
      25 35 45 55 65
```

The system displays:

```
TOTAL SALES 2317.5
```

When you try to display *SALES*, you still get a *VALUE ERROR*, because *SALES* still does not exist in the workspace.

```
      SALES
VALUE ERROR
      SALES
      ^
```

Summary

In this chapter, you learned:

- You can collect numeric input using evaluated input or quad input.
- You can collect character input using character input or quote-quad input.
- You can use the branch function (→) for looping. Functions can branch to a line number or a line label. Labels are more meaningful and do not change if line numbers change.
- You can include comments in a function to summarize in words what the function does. You precede comments by a lamp symbol (⍝) so the system does not interpret them as part of the function.
- Local variables are variables that have meaning only inside functions. You can use local variables to prevent functions from changing your general workspace variables (global variables) and also save room in the workspace.
- You can use functions to collect data, store data in files, and retrieve data from files.

Exercises

A. 1. Write a new version of *DIE* (from the Writing Programs in APL chapter). Call the new version *ASKDIE*. *ASKDIE* rolls the die, then gives you another chance. The function should produce the following display:

```
      ASKDIE
5
TRY AGAIN?  (YES OR NO):
YES
2
TRY AGAIN?  (YES OR NO):
NO
```

2. Write the function *ADDER*, which produces the following display:

```
      ADDER
ENTER NUMBERS TO ADD:
⎕:
      107 542 122 889
THE TOTAL IS 1660.
```

3. Define a function called *ASKREPEAT* that repeats the character you enter. *ASKREPEAT* should produce the following display:

```
      ASKREPEAT
ENTER THING TO REPEAT:
*
REPEAT HOW MANY TIMES?
⎕:
      13
*************
```

4. Copy the *HIST* function from the Writing Programs in APL chapter into the active workspace. Use

```
      )COPY MYWORK HIST
```

If you did not save the function, copy it in from *LESSONS*. Use *HIST* inside a new function called *ASKHIST*. *ASKHIST* should produce the following display:

```
      ASKHIST
ENTER FOUR NUMBERS FOR CHART:
⎕:
      6 12 3 1
⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
⎕⎕⎕
⎕
```

8 Formatting

In this chapter, you will learn:

- how to format integers, floating-point numbers, and character data using the format function (⍕) and the system function ⎕*FMT*
- how to add decorations such as commas, dollar signs, and percent symbols to formatted data
- how to use several different column formats in one report.

What Is Formatting?

APL★PLUS displays data in certain ways. This display of data is called *formatting*. You can control formatting using two tools: the format function (⍕) and the system function ⎕*FMT*.

You have already used the format function to change numeric data into character data. You can create even more elaborate formats when you use format dyadically; that is, with a left and a right argument.

Dyadic Format

Just like the monadic format function, the dyadic format function turns numeric data in the right argument into character data. The difference is that the left argument tells the dyadic format function how to arrange the data.

format pairs

The left argument consists of one or more pairs of numbers called format pairs. The first number of a format pair specifies the field width; that is, the number of spaces in each column. The second number of a format pair specifies how to format the data; that is, the number of digits to display to the right of the decimal point. You can use one format pair to format all of your data the same way, or you can use several pairs to format each column of data differently. (A *column* is a single number or letter (a scalar), each element of a vector, or each column of a matrix.)

Try formatting the following vector with a field width of 6 and no digits to the right of the decimal point. Enter:

```
      DATA←1032 10.52 ¯20.11 3.1 2 ¯50
      6 0 ⍕ DATA
```

The system displays:

```
  1032    11   ¯20     3     2   ¯50
```

Formatting Whole Numbers

The system used the format pair, 6 0, to format each column, or element, in *DATA*. The first number, 6, tells the system to make each column six spaces wide. The second number, 0, specifies how many digits to the right of the decimal point to display. Notice that the system did not display the numbers to the right of the decimal place.

The system displays:

```
  1032      11     ¯20
     3       2     ¯50
```

Now, try formatting each column differently:

```
      6 0 6 2 7 2 ⍕ DATA
```

The system displays:

```
  1032 10.52 ¯20.11
     3  2.00 ¯50.00
```

The first column has whole numbers and a field width of six; the second column has two decimal places and a field width of six; and the third column also has two decimal places, but a field width of seven.

⎕*FMT*

Dyadic format gives you more control over the way you display data. But, you may want more types of formatting than just controlling the placement of the data and the number of decimal places — especially for reports. For example, you might want commas, dollar signs, or decimal points for dollar amounts. You can do all of this with the system function ⎕*FMT*.

Format Phrases

⎕*FMT* uses format phrases surrounded by single quotes as its left argument. Its right argument is data (scalars, vectors, or matrices). The format phrase consists of codes that tell ⎕*FMT* how to format the data in the right argument. They specify the type of data (character or numeric) and the field width. (Remember that the field width is the number of spaces allowed for each column.)

Common types of numeric formatting are integer (whole number) and floating point (numbers with decimal points). ⎕*FMT* uses the *I* format phrase to format integers and the *F* format phrase to format floating-point numbers.

To specify the field width, type a number to the right of the *I* or *F*. For example, *I*6 displays each integer in a column six spaces wide:

```
      'I6' ⎕FMT DATA
  1032    11   ¯20
     3     2   ¯50
```

You can use as many of these format phrases together as you like. Remember, each decoration counts as a space, so you must allow enough spaces in each column. If you do not, the system displays stars. Try the following examples:

```
      'P<$>F8.2' ⎕FMT 1082.2
```

The *text* in this case is a dollar sign. The system displays:

```
$1082.20
```

The next example uses two decorations. Enter:

```
      'M<($>N<)>F10.2' ⎕FMT ¯1082.2
```

In this case, the *M* puts a left parenthesis and dollar sign on the left side of the `¯1082.2`. The *N* puts a right parenthesis on the right side of the `¯1082.2`. Therefore, the system displays:

```
($1082.20)
```

The next example uses different decorations for positive and negative numbers. Enter:

```
      'P<$>Q< >M<($>N<)>F15.2' ⎕FMT 1002.6
345.32 ¯214.5
```

The *P* puts a dollar sign on the left of the positive numbers. The *Q* puts a space on the right of the positive numbers. The *M* puts a left parenthesis and a dollar sign on the left of the negative numbers. The *N* puts a right parenthesis on the right of the negative number. Therefore, the system displays:

```
      $1002.60
       $345.32
      ($214.50)
```

Notice in the last example that ⎕*FMT* stacks vectors (displays them as a single column of numbers).

To display percentage signs on the right of some numbers, use a percent sign as the decoration; for example:

```
      'N<%>Q<%>F6.1' ⎕FMT 31.2 87.5 ¯23.8
 31.2%
 87.5%
¯23.8%
```

You can flag numbers with any text. For example, to insert the letters *DR* and *CR* (for debit and credit) next to positive and negative numbers, enter:

```
      'M<>N< CR>Q< DR>F9.2' ⎕FMT 32.34
¯21.4
```

The system displays:

```
32.24 DR
21.40 CR
```

The *M*<> keeps the system from printing the negative sign.

Formatting and Variables

You can assign format strings to variables, so that you can use them again and again. Enter the following example:

```
      FS←'N<%>Q<%>F8.1'
      FS ⎕FMT 31.2 87.5 ¯23.8
   31.2%
   87.5%
  ¯23.8%
```

Formatting Individual Columns

To format each column of a matrix differently, set up the format phrase for each column, then join the individual format phrases with commas. The following example uses three different format phrases, separated by commas to format three columns of data. The three format phrases are stored in the variable *FS*1.

```
      DATA
1032   10.52 ¯20.11
   3.1  2    ¯50

      FS1←'M<($>N<)>P<$>F9.2,N<%>Q<%>
I5,M<>N<CR>F10.2'
      FS1 ⎕FMT DATA
 $1032.00  11%  20.11 CR
    $3.10   2%  50.00 CR
```

The three format phrases format the data as follows:

- *M*<($>*N*<)>*P*<$>*F*9.2 puts a left parenthesis and a dollar sign on the left of negative numbers, a right parenthesis on the right of negative numbers, and a dollar sign on the left of positive numbers. The phrase specifies a field width of nine and two decimal places.

- *N*<%>*Q*<%>*I*5 puts a percent sign on the right of negative and positive numbers in a field width of five.

- *M*<>*N*<*CR*>*F*10.2 suppresses the high minus sign in negative numbers and puts the letters *CR* on the right side of negative numbers. The phrase specifies a field width of 10 and two decimal places.

What if you only used two phrases? Enter:

```
      'I8,F8.2' ⎕FMT DATA
```

The system displays:

```
    1032   10.52     ¯20
       3    2.00     ¯50
```

⎕FMT formats columns 1 and 2 with the phrases you entered. Then, *⎕FMT* goes back to the beginning of the format string and repeats the first phrase to format column 3. If *DATA* contained more columns, the system would repeat the format phrases until all data columns were formatted.

Repetition Factor

In the preceding example, you saw that *⎕FMT* repeats format phrases if you do not enter a phrase for each data column. You can also use a repetition factor to repeat format phrases. A repetition factor allows you to control the phrases *⎕FMT* repeats and how many times *⎕FMT* repeats them. Enter the repetition factor *before* the format phrase; for example:

```
'I8,2F8.2' ⎕FMT DATA
```

The system displays:

```
1032    10.52    ¯20.11
   3     2.00    ¯50.11
```

Since you put the 2 before the *F*, *⎕FMT* formatted the second and third columns with the *F*8 . 2 phrase.

Formatting Multiple Pieces of Data

⎕FMT can format many different pieces of data at once. Enclose each piece of data in parentheses and separate the pieces from each other by semicolons (;). For example, enter:

```
NUMS←35.7 20.11
'2I8,2F8.2' ⎕FMT (NUMS;DATA)
```

The system stacks and formats the vector *NUMS* next to *DATA*:

```
36      1032    10.52  ¯20.11
20         3     2.00  ¯50.00
```

You can also format character data and numeric data at the same time. To format a character matrix, use the *A* format phrase with a repetition factor equal to the number of columns in the character matrix. Enter:

```
      MONTHS←2 3ρ'JANFEB'
      '3A1' ⎕FMT MONTHS
```

⎕*FMT* allocates one column for each letter and repeats this allocation three times:

```
JAN
FEB
```

When you format character and numeric data together, be sure the repetition factors are right, since you cannot use the *A* format phrase with numbers. The following example shows how to format character data and numeric data.

```
      '3A1,3F8.2' ⎕FMT (MONTHS;DATA)
JAN 1032.00   10.52  ¯20.11
FEB    3.10    2.00  ¯50.00
```

Basic Reporting Functions

You can use formatting inside a function to produce a report. Create the following variables.

```
      INCOME←3 3ρ10520.2 11500 12500
11217.71 11000 11500 0 0 0

      NAMES←3 7ρ'REVENUEEXPENSEPROFIT '
```

Display the variables:

```
      INCOME
10520.2  11500    12500
11217.71 11000    11500
    0        0        0

      NAMES
REVENUE
EXPENSE
PROFIT
```

Create the functions shown below. The first function, *CALC*, calculates profit by subtracting expenses (in row 2 of *INCOME*) from revenue (in row 1 of *INCOME*). Then it assigns the answer to row 3 of *INCOME* (where you left zeros).

The second function, *REPORT*, prints and centers the headings, runs the *CALC* function, and then prints the body of the report. To insert blank lines in the report, enter a single blank (' ') on the appropriate lines of the *REPORT* function.

```
      ∇CALC
[1]   INCOME[3;]←INCOME[1;]-INCOME[2;]
[2]   ∇

      ∇REPORT
[1]   '                  HOMEGROWN CO.'
[2]   ' '
[3]   '           1989        1990        1991'
[4]   ' '
[5]   CALC
[6]   '7A1,3CF12.2' ⎕FMT (NAMES;INCOME)
[7]     ∇
```

Now run the *REPORT* function:

```
       REPORT
```

The system prints the report:

```
                  HOMEGROWN CO.

              1989        1990        1991

REVENUE  10,520.20   11,500.00   12,500.00
EXPENSE  11,217.71   11,000.00   11,500.00
PROFIT     ¯697.51      500.00    1,000.00
```

You can edit the *REPORT* function to add dollar signs, parentheses, and anything else you want to include.

Summary

In this chapter, you learned:

- how to use the dyadic format function (⍕)
- how to use the system function ⎕*FMT*.

Table 8-1 shows the format phrases you learned to use. ⎕*FMT* offers many more capabilities in addition to those introduced here. Refer to your system documentation for more details.

Table 8-1. Format Phrases

Format Phrase	Description
r Aw	Formats character data.
C	Inserts commas for thousands, millions, and so on.
r Fw.d	Formats floating-point data (numbers with decimal points).
r Iw	Formats integers (whole numbers).
M <text>	Places text inside < > on the left side of negative numbers.
N <text>	Places text inside < > on the right side of negative numbers.
P <text>	Places text inside < > on the left side of positive numbers.
Q <text>	Places text inside < > on the right side of positive numbers.

Key: *r* = repetition factor
w = width (number of columns)
d = number of decimal places.

Exercises

A. Create the following variable:

```
          D←2 3⍴10250.5 17512.7 18016.2
   11205.81 16867.12 19197.2
```

Use *D* with ⎕*FMT* to do the following exercises.

1. Format the numbers in variable *D* as integers, using a field width of eight.

2. Format *D* as floating-point numbers, using a field width of ten and two decimal places.

3. Format *D* as floating-point numbers, using a field width of eight and two decimal places.

4. Format *D* as floating-point numbers, using a field width of 18 and two decimal places.

5. Repeat Exercise 4, using the *C* modifier to insert commas in the data.

6. Repeat Exercise 5, using the *P* modifier to place dollar signs next to the numbers.

B. Create the variables *E* and *N*:

```
E←2 2⍴18147.7 20.4 19717.11 ¯17.3
N←2 8⍴'REVENUESEXPENSES'
```

1. Format column 1 of *E* as floating-point numbers, with a column width of 12 and two decimal places. Format column two of *E* as floating-point, with a column width of 10 and one decimal place.

2. Repeat Exercise 1, inserting commas and placing dollar signs next to the numbers in column 1.

3. Repeat Exercise 2, placing percent signs (%) on the right of the numbers in column 2.

4. Repeat Exercise 3, changing the APL high minus (the one next to the negative percentage) to a hyphen.

5. Repeat Exercise 4, placing a plus sign (+) next to the positive percentage.

6. Repeat Exercise 5, formatting *N* next to *E*. That is, format *N* first, then *E*, using *one* ⎕*FMT* statement. The result should look like this:

```
REVENUES  $18,147.70  +20.4%
EXPENSES  $19,717.11  -17.3%
```

C. Create the variables *S* and *N*1:

```
      S←2 2⍴6543.1 5342.55 6231.89 7531.21
      N1←2 2⍴'R1R2'
```

1. Format the numbers in *S* as floating point, with a column width of 12 and two decimal places.

2. *S* contains the sales figures for two sales representatives for two weeks. The rows correspond to the sales representatives; the columns to the weeks. You can calculate the total sales for two weeks with plus reduction:

```
      T←+/S
      T
11885.65 13763.1
```

   Repeat Exercise 1, formatting the totals (*T*) next to *S* (on the right-hand side of *S*). Use the same format phrase for both pieces of data.

3. Repeat Exercise 2, inserting dollar signs next to all the numbers.

4. Repeat Exercise 3, inserting commas in the thousands position.

5. Format *N*1 on the left of the numbers. (*HINT*: Remember to use repetition factors for *both* format phrases.)

D. 1. Define a function *REPT* that produces a report using the information from Exercise C.5. The function should have the syntax *REPT data*. *REPT* finds row totals for *data* using +/, stores the totals in a variable called *TOT*, then formats *N*1, *data*, and *TOT* together. Test your function on *S* by entering:

```
        REPT S
```

2. Define a new function *RPT*2 that behaves the same as *REPT*, except that it prints the centered title *SALES FIGURES* and the column titles *WEEK*1, *WEEK*2, and *TOTALS* before it formats the body of the report.

9

Storage Facilities: Workspaces

In this chapter, you will learn:

- what a workspace is
- how to store functions and variables in a workspace
- how to display a list of the workspaces on your system
- how to change the name of a workspace
- how to erase individual functions and variables in the workspace
- how to erase a workspace.

What Is a Workspace?

A workspace is a place in the computer where APL★PLUS performs work and stores objects (functions and variables). You have been working in workspaces all this time — either in "clear" ones or in ones you have named. Think of workspaces as different folders in a file drawer. When you want to use the documents in a file folder, you locate and retrieve the folder from the drawer. When you are finished, you must replace the documents in the folder or they will be lost.

Note: On most APL systems, a logical collection of related workspaces is called a library; each library has a different library number. On some APL systems, you can use disk directory names as well as library numbers. If you omit the library number when you type the name of a workspace, APL assumes you want a workspace in the current library or on the current disk directory. To find out how your APL★PLUS System manages libraries or disks, refer to your system documentation.

active workspace

Before you can use the functions and data in a workspace, you must activate or load it using the command *)LOAD*. To store a workspace that contains new or revised items, use the command *)SAVE*. You can move back and forth between workspaces using these two commands. You load one workspace, save it when you are finished, then load another, save it, and so on.

Whenever you are in APL, you are *always* using a workspace. You call the workspace you are in the active workspace (as opposed to stored workspaces). The active workspace may be a clear one that has no name, or it may already have a name. Most systems give you a clear workspace when you sign on. You can always give it a name by storing it under a name you choose. For example, enter:

```
)SAVE MYWORK
```

APL★PLUS displays a message telling you name of the workspace name and that it has been stored. The exact wording of this message may vary from system to system.

No matter which workspace is active, all of the built-in APL functions (+, ÷, +/, ρ, and so on) are always available for you to use.

System Commands

Besides saving and loading workspaces, you can do other things with workspaces, such as listing, erasing, or copying the variables and functions they contain. The commands that allow you to perform these actions are called system commands. These commands control the system rather than performing calculations. All system commands begin with a right parenthesis.

Saving and Loading Workspaces

saving a workspace

The system command *)SAVE* stores a copy of the active workspace using the name you specify. It takes the form:

```
)SAVE  name
```

Enter:

```
)SAVE  EXAMPLE
```

APL displays a message telling you when the workspace was stored. If you try to save a clear workspace, APL displays an error message. Be sure to name the clear workspace when you save it.

Notice that a *copy* of the workspace is stored. When you enter *)SAVE*, the active workspace remains the same.

If the workspace already has a name, you only need to enter:

```
)SAVE
```

The system saves the workspace under its old name. The *SAVED* message includes the name of the workspace, which is the name you gave it a moment ago — *EXAMPLE*.

Remember, the system does not save the variables and functions you create in the active workspace until you use the `)SAVE` command.

loading a workspace

To activate a stored workspace, use the `)LOAD` command . It takes the form:

```
          )LOAD  name
```

Try loading the *DEMOAPL* workspace:

```
        )LOAD  DEMOAPL
```

The system displays a *SAVED* message that tells you the last time the workspace was saved; for example:

```
SAVED  01/18/91   13:16:27
```

If the *DEMOAPL* workspace is not in the current library or on the current disk, type the appropriate library number or disk directory name, depending on your APL system.

Remember, although you have loaded the workspace, APL★PLUS does not store any of the things you create in it until you use the `)SAVE` command.

Clearing the Workspace

The `)CLEAR` command removes the current workspace from the computer's memory and returns you to a clear workspace. Clear the workspace before you create functions and variables that you want to keep separate from other workspaces.

For example, define two variables, *A* and *B*, and create the *ADD* function shown below:

```
        A←5
        B←⍳9
        ∇X ADD Y
[1]     X+Y
[2]     ∇
```

Save the workspace under the name *EXAMPLE*:

```
          )SAVE  EXAMPLE
```

Then, clear the workspace:

```
          )CLEAR
```

The system displays a message indicating the clear workspace.

```
CLEAR WS
```

Now enter:

```
          A
```

The system displays:

```
VALUE  ERROR
       A
       ^
```

Enter:

```
       7  ADD  8
```

The system displays:

```
SYNTAX  ERROR
       7  ADD  8
          ^
```

Since the *EXAMPLE* workspace is no longer active, you cannot use *A*, *B*, and *ADD*.

Listing Functions and Variables

The)*FNS* command lists the names of the functions in the workspace in alphabetical order. The)*VARS* command lists the names of the variables in the workspace in alphabetical order. To see how they work, load the *EXAMPLE* workspace:

```
          )LOAD  EXAMPLE
```

Enter:

```
      )FNS
```

The system displays the names of the functions in the workspace:

```
ADD
```

Enter:

```
      )VARS
```

The system displays the names of the variables in the workspace:

```
A          B
```

Erasing Functions and Variables

The)*ERASE* command deletes variables and functions from the workspace. It takes the form:

```
      )ERASE  object1 object2 . . .
```

Enter:

```
      )ERASE  A
```

The system displays a blank line with the cursor indented six spaces. Now, list the variables:

```
      )VARS
B
```

You can see that *A* no longer exists in the active workspace. The deletions you make are in the active workspace and do not affect the stored copy of the workspace, unless you enter `)SAVE` after deleting an object. For example, load *EXAMPLE* again and list the variables:

```
       )LOAD EXAMPLE
SAVED . . .

       )VARS
A        B
```

To erase an object permanently, use `)ERASE` to erase the object; then use `)SAVE` afterward.

Listing Workspaces

The commands `)WSLIB` or `)LIB` (depending on your APL system) list the workspaces you have in a library. Enter:

```
        )WSLIB
```

The system displays the names of the saved workspaces in your current library. Enter

```
       )WSLIB 2
```

to list the workspaces in library 2. The list of workspaces you see on your screen depends on the workspaces you have stored; however, the list should include *EXAMPLE*.

Identifying the Active Workspace

The `)WSID` command displays the name of the active workspace. Try entering:

```
       )WSID
```

The system displays:

```
IS EXAMPLE
```

What happens when you clear the workspace?

```
      )CLEAR
CLEAR WS

      )WSID
IS CLEAR WS
```

You can also use the)*WSID* command to rename the active workspace:

```
      )WSID EX1
WAS CLEAR WS

      )WSID
IS EX1
```

After you rename the workspace, you can use)*SAVE* to save it.

Deleting Workspaces

The)*DROP* command deletes a stored workspace. It has the form:

```
      )DROP name
```

The system drops the specified workspace from storage. This command does *not* affect the active workspace. For example, define the following variable in the workspace *EX*1.

```
      XX←⍳11
```

Save *EX*1:

```
      )SAVE
SAVED . . .
```

Enter *)LIB* or *)WSLIB*. Notice that the system adds *EX*1 to your list of stored workspaces. Now, drop *EX*1:

```
      )DROP EX1
```

If you enter *)WSLIB* now, you notice the *EX*1 no longer appears in the list; however, the active workspace is still *EX*1:

```
      )WSID
IS EX1
```

Also, notice that the variable *XX* still exists:

```
      XX
1 2 3 4 5 6 7 8 9 10 11
```

If you load another workspace or clear the workspace without saving it first, *EX*1 disappears. Enter:

```
      )CLEAR
CLEAR WS

      )LOAD EX1
WS NOT FOUND
```

Copying Objects

The *)COPY* command copies objects (functions and variables) from one workspace to another. It takes the form:

```
      )COPY wsname objectnames
```

In the statement shown above, *wsname* stands for the workspace name, and *objectnames* is a list of the objects you want to copy. For example, to copy the *SALES* variable from the *MYWORK* workspace, enter:

```
      )COPY MYWORK SALES
SAVED . . .
```

Check to see if *SALES* exists in the active workspace:

```
      SALES
125 350 201 115 279
```

Your values for *SALES* may be different. If you do not have a *MYWORK* workspace, APL displays the message *WS NOT FOUND*. Try the example above with any object stored in one of the workspaces listed by)*WSLIB*.

If you omit the *objectnames* when you enter the)*COPY* command, APL copies the entire workspace into your active workspace. Enter:

```
      )COPY MYWORK
SAVED . . .
```

If you list the variables now, you see all the functions and variables from *MYWORK*.

The)*COPY* command only brings things into your active workspace. To store new objects permanently, use the)*SAVE* command.

Signing Off

The)*OFF* command signs you off from the APL system. (On APL★PLUS, be sure to use this command to protect any active files. You will learn about files in the Storage Facilities: Files chapter.)

Table 9-1. Continued

Command	Description
`)WSID`	Shows the name of the active workspace.
`)WSID wsname`	Changes the name of the active workspace to the name indicated.
`)WSLIB` or `)LIB`	Lists the workspaces in a library.

This chapter has no formal exercises. Using your stored workspaces, practice the commands you learned; however, use caution with the `)DROP` command.

10

Storage Facilities: Files

In this chapter, you will learn:

- how to create files
- how to store and retrieve data from your files
- how to display a list of the files in a library or directory
- how to erase files when you no longer need them.

What Is a File?

Files provide data storage outside of workspaces. A difference between files and workspaces is that files can store only data, while workspaces can store both functions and data. You cannot perform computations inside files. To use the data in a file, you must bring a copy of the data into your active workspace, work with it, and then put it back in the file.

If you think of the active workspace as a folder in a desk drawer, you can think of a file as a filing cabinet in the same room. If you want a folder from your desk, you have it right at hand. But if you want to work on a folder located in the filing cabinet, you must get up and get it. And, if you want to make sure that you do not lose the documents in the folder, you must put the folder back in the filing cabinet when you are finished.

You may not need a filing cabinet if you do not have many things to store. Your desk has a lot of room for storing things. But if you need to store a lot of data, you need a filing cabinet.

It is the same in APL★PLUS. If you do not have much to store, you can use the workspace as storage. But if you have a lot of data, it is more convenient to store it in a file. Files are also useful for organizing data.

Files have many advantages over workspaces. The main ones are:

- Files can hold much more data than a workspace.
- Files remain active when you change workspaces.
- Files give you extra storage without using any room from the workspace.

APL★PLUS does not automatically provide files for you — you must create them. Files consist of discrete parts called components. Each component can contain something different, as variables can. When you first create a file, it has no components; that is, it is empty.

You add components to the file by appending them to the end of the file. Components can contain numeric or character data of any size or shape — from single numbers to matrices or multi-dimensional data. Some components can even be empty.

Rather than names, components have numbers. The system numbers the components sequentially. This behavior makes it convenient to refer to the components by their numbers.

You change the data in file components by replacing the component. The new data can be slightly different than the old data or completely different . For example, you can catenate a single number to a vector and replace the vector, or you can replace a numeric vector with a character matrix.

To use a file, you must first activate the file by associating the file with a number. The file is tied to the number. You can then refer to the entire file by that number. For example, to read the first component of a file tied to the number 33, enter:

```
      ⎕FREAD 33 1
```

The 33 is the tie number of the file; that is, the number you used to activate the file. The 1 is the number of the component you want to read.

File Functions

All system functions that operate on files begin with the characters ⎕*F*. The ⎕ means that they are system functions, and the *F* stands for file. There are approximately two dozen system functions you can use with files; this chapter highlights the simplest and most useful ones.

Creating a File

You can create a file using the command ⎕*FCREATE*. It has the format:

```
      'name' ⎕FCREATE tie number
```

The *name* is the name you want to give the new file. Since the file name is character data, you must surround it with single quotes. The name can contain both letters and numbers, but it must begin with a letter. Different APL★PLUS systems allow different lengths for file names; refer to your system documentation. The *tie number* is any positive whole number you choose.

For example, to create a file named *TEST* with a tie number of 1, enter:

```
      'TEST' ⎕FCREATE 1
```

If you try to create another file with the same name, ⎕*FCREATE* displays an error message; for example, if you enter

```
      'TEST' ⎕FCREATE 2
```

the system displays:

```
FILE NAME ERROR
      'TEST' ⎕FCREATE 2
             ^
```

Adding Data to a File

The ⎕*FAPPEND* function lets you add data to a file. It has the format:

data ⎕*FAPPEND* *tie number*

The data you specify can be either numbers, character data, or the name of a variable that contains numbers or characters. If you want to append character data, you must enclose it in single quotes.

For example, create and display the following variable:

```
      VAR←ι5
      VAR
1 2 3 4 5
```

Now, append it to the file:

```
      VAR ⎕FAPPEND 1
```

The system displays:

```
1
```

This number is the number of the component . (Since this number varies depending on how many components you add, the following examples omit this number.)

Append some more data to the *TEST* file:

```
      22 ⎕FAPPEND 1
      'THIS IS A TEST.' ⎕FAPPEND 1
      DATA←3 3ρ'ONETWO3  '
      DATA ⎕FAPPEND 1
```

Checking the Size of a File

The ⎕*FSIZE* function lets you check the size of a file. It has the format:

```
      ⎕FSIZE tie number
```

Its result displays the following information:

- The starting component number of the file (usually 1).
- The number of the component that the system will append next.
- The size of the file in bytes.
- The growth limit for the file. A 0 means "no limit" — the file can grow to virtually any length (that is, until the computer you are working with runs out of storage space on your hard disk or floppy-disk drive).

(Some APL systems include additional elements in the result. For more information, refer to the documentation for your APL system.)

To check the size of the file you just created, enter:

```
      ⎕FSIZE 1
```

The system displays the file information; for example:

```
1 5 512 0
```

The third and fourth items vary depending on the system you are using. Note that second item shows the next component the system will fill. Since the size of the file you created is 1 5, you know that components 1 through 4 have already been used, and the next component you append will be number 5.

Reading Data from a File

The ⎕*FREAD* function lets you read the data in a file. It has the following general form:

```
      ⎕FREAD tie number component number
```

For example, to read the data in your file, enter:

```
      ⎕FREAD 1 3
```

The first number in the expression, 1, is the tie number of the file. The second number, 3, is the component whose data you want to read. The system displays the data stored in that component:

```
THIS IS A TEST.
```

You can store the result in a variable for further work. The next example stores the contents of component 3 in the variable *S*:

```
      S←⎕FREAD 1 3
      S
THIS IS A TEST.
```

The next example deletes the blanks from *S*.

```
      (S≠' ')/S
THISISATEST.
```

The next example changes the value of *S*:

```
      S←4⍴S
      S
THIS
```

Any work you do has no affect on the component in the file:

```
      ⎕FREAD 1 3
THIS IS A TEST.
```

If you try to read a component that does not exist, APL displays an error message; for example, if you enter

```
      ⎕FREAD 1 72
```

the system displays:

```
FILE INDEX ERROR
      ⎕FREAD 1 72
      ^
```

Replacing Components

To change a component in a file, use the ⎕*FREPLACE* function. This function has the format:

data ⎕*FREPLACE* *tie number component number*

For example, to change the data stored in component 3 of *TEST*, enter:

```
      'TEST NUMBER TWO' ⎕FREPLACE 1 3
```

If you read component 3 now, you see that the system replaced it:

```
      ⎕FREAD 1 3
TEST NUMBER TWO
```

Untying a File

To deactivate a file, use ⎕*FUNTIE*, which unties the file. ⎕*FUNTIE* breaks the association between a file name and a tie number. It takes the form:

⎕*FUNTIE* *tie number(s)*

For example, to untie the *TEST* file, enter:

```
      ⎕FUNTIE 1
```

You can no longer read the file now:

```
      ⎕FREAD 1 3
FILE TIE ERROR
      ⎕FREAD 1 3
      ^
```

Be sure to untie all files before removing a disk or resetting the system. On all systems, the *)OFF* command automatically unties any files currently tied.

Tying a File

To activate a file that you have already created, use the *⎕FTIE* function to tie it. *⎕FTIE* takes the form:

```
      'name' ⎕FTIE tie number
```

For example, to tie the *TEST* file, enter:

```
      'TEST' ⎕FTIE 7
```

You can use any whole number as a tie number, as long as the system is not using it for another file. (If the system is already using the number, a *FILE TIE ERROR* occurs.) Once you activate a file, you can read data the data it contains; for example:

```
      ⎕FREAD 7 3
TEST NUMBER TWO
```

Tie numbers are only temporary identifiers for a particular file. You can use a different tie number each time you tie the file. But if you tie more than one file at the same time, you must use a different tie number for each one. If two different files were tied to the same number, the system would not know which file to read when you used *⎕FREAD*.

Identifying Files

To display the names of the files you currently have tied, use the ⎕*FNAMES* function. ⎕*FNAMES* takes no arguments; that is you enter:

```
      ⎕FNAMES
```

The system displays the files you currently have tied; for example:

```
TEST
```

Depending on your APL system, a library number or disk directory name may appear before the file name.

To display the numbers your files are tied to, use the ⎕*FNUMS* function. ⎕*FNUMS* takes no arguments; that is you enter:

```
      ⎕FNUMS
```

The system displays the tie numbers you are currently using; for example:

```
7
```

If you have more than one file tied, ⎕*FNUMS* displays a vector of tie numbers. The order of the vector corresponds to the order of the file names produced by ⎕*FNAMES*.

Because the result of ⎕*FNUMS* contains the tie numbers of all of your tied files, you can untie all tied files at once by entering:

```
      ⎕FUNTIE ⎕FNUMS
```

Erasing Files

The *⎕FERASE* function lets you erase a file you no longer need. It has the format:

```
        'name' ⎕FERASE  tie number
```

You must tie a file before you can erase it. You must enter both the file name and its tie number to make sure you erase the correct file. You must also supply the library number or disk name before the file name.

For example, to erase the file *TEST*, enter:

```
        'TEST' ⎕FERASE 7
```

⎕FNAMES now shows that no files are tied:

```
        ⎕FNAMES
```

Create one more file. Enter:

```
        'TEST' ⎕FCREATE 22
```

Notice that the new file can have the same name as the old file, since the old one no longer exists. Untie this file. Enter:

```
        ⎕FUNTIE 22
```

⎕FLIB lists the files in a particular library or directory. It has the form:

```
        ⎕FLIB  identifier
```

The argument *identifier* is the library number or directory name whose contents you want to list. For example, on APL★PLUS, to list the files in disk drive C, you enter:

```
        ⎕FLIB 2
```

The system displays:

```
2 TEST
```

The system may display additional file names, depending on the disk.

On APL★PLUS systems, the system command *)FLIB* lists files.

```
      )FLIB 2
2 TEST
```

Functions and Files

You can use what you have learned about programming to build some useful functions that work with files.

For example, you might want to build a function that creates a file containing your favorite recipes. Each component in the file contains a single recipe.

building a recipe

A recipe is a good example of a character matrix. You can create the character matrix using the shape function (⍴) and store the matrix in a variable; for example:

```
      RECIPE←3 12⍴'1 CUP FLOUR PINCH SALT
1 EGG'
```

Display the variable:

```
      RECIPE
1 CUP FLOUR
PINCH SALT
1 EGG
```

using a function to build a recipe

Although this method works, it is cumbersome to create a recipe this way. It is easier to have a function build the matrix line by line, using the catenation function (,) to add each additional line; for example:

```
      NEWLINE←1 12⍴'BAKING PWDR '
      RECIPE,[1]NEWLINE
1 CUP FLOUR
PINCH SALT
1 EGG
BAKING PWDR
```

The function should assemble a recipe in this way, then add it to the file. You also have to build in a word like *END* or *STOP* so the function knows where the recipe ends.

The following function, *FILERECIPE*, illustrates these techniques.

```
     ∇ FILERECIPE;RECIPE;NEWLINE
[1]    ⍝ COLLECTS AND FILES RECIPE
[2]     'RECIPES' ⎕FTIE 9971
[3]     'ENTER YOUR RECIPE LINE BY LINE'
[4]     '(USE ¨END¨ TO STOP):'
[5]     RECIPE←60↑⍞
[6]     RECIPE← 1 60 ⍴RECIPE
[7]    L1:'NEXT LINE:'
[8]     NEWLINE←60↑⍞
[9]     →(∧/'END'=3↑NEWLINE)⍴L2
[10]    RECIPE←RECIPE,[1]NEWLINE
[11]    →L1
[12]   L2:RECIPE ⎕FAPPEND 9971
[13]    ⎕FUNTIE 9971
[14]    'DONE.  NEW RECIPE FILED.'
[15]   ∇
```

This function ties the file for you, then collects the text for the recipe. When you enter *END*, the system stores the recipe. The tie number should be one that you are unlikely to use to tie any other files you might be working with, because file tie numbers used at the same time must be unique. *FILERECIPE* then constructs the recipe as follows.

- Lines `[5]` and `[6]` gather the first line of the recipe and convert the line into a one-row matrix (so the system can catenate the line using matrix techniques).

- Lines `[7]` and `[8]` gather the next recipe line.

- Line `[9]` checks to see if the first three characters are the word *END*.

- If not, the system goes on to line `[10]`, which catenates the line to the end of the recipe.

- Line `[11]` returns to line `[7]` for the next line of the recipe.

making sure the lines are equal

Remember, every line in the recipe must be of equal length so that you can catenate them. You use the take function (60↑⍞) to make sure each line is of equal length. Take pads each line of the recipe with blanks. When you end the function, you have a recipe in the form of a matrix with 60 columns and as many rows as the recipe has lines.

At the end of the function, the system unties the file again. Notice that the function assumes that you have already created a file called *RECIPES* before you begin the function.

Create the file *RECIPES*.

```
      'RECIPES' ⎕FCREATE 1
```

Then untie it.

```
      ⎕FUNTIE 1
```

Now run the function.

```
ENTER YOUR RECIPE LINE BY LINE
(USE "END" TO STOP):
COOKIES
NEXT LINE:
```

(Press the Return or Enter keys to generate a blank line.)

```
NEXT LINE:
GO TO GROCERY STORE.
NEXT LINE:
BUY A BAG OF YOUR FAVORITE COOKIES.
NEXT LINE:
TAKE HOME, OPEN, AND EAT.
NEXT LINE:
END
DONE.  NEW RECIPE FILED.
```

To read the recipes, use ⎕*FTIE* to tie the file and ⎕*FREAD* to read a component; for example:

```
      'RECIPES' ⎕FTIE 1
      ⎕FREAD 1 1
COOKIES
GO TO GROCERY STORE.
BUY A BAG OF YOUR FAVORITE COOKIES.
TAKE HOME, OPEN, AND EAT.
```

using a function to read a recipe

To read a recipe without using ⎕*FREAD* directly, you can create a function that reads the file, like the *READ* function in the next example.

```
      ∇ READ;ANS
[1]    ⍝ READS RECIPES STORED IN FILE
[2]     'RECIPES' ⎕FTIE 3371
[3]     'WHAT IS THE RECIPE NUMBER?'
[4]     ANS←⎕
[5]     ⎕FREAD 3371,ANS
[6]     ⎕FUNTIE 3371
[7]   ∇
```

Now, run the *READ* function. Enter:

```
      READ
```

The system displays:

```
WHAT IS THE RECIPE NUMBER?
⎕:
```

Enter the component number. Because you have only filed one recipe, enter:

```
      1
```

The system displays:

```
COOKIES
GO TO GROCERY STORE.
BUY A BAG OF YOUR FAVORITE COOKIES.
TAKE HOME, OPEN, AND EAT.
```

using a file directory

If you cannot remember which component has which recipe, you can add a function to *FILERECIPE* that creates a directory of the stored recipes. To build a directory, take the first line of each recipe (the title) and store it in a separate matrix in the file.

In fact, you can even build a system with small functions that perform each job — tying the file, keeping the directory, and so on. You use a main function (also called a driver function) to run all of the smaller functions.

using a driver function

The main function would look like the following example, called *RECIPES*.

```
      ∇ RECIPES
[1]    ⍝ MANAGES RECIPE SYSTEM
[2]     TIEFILES
[3]    L0:'LIST, READ, ADD, OR QUIT?'
[4]     '(L, R, A, OR Q):'
[5]     →('LRAQ'=1↑⍞)/L1,L2,L3,L4
[6]     'PLEASE ENTER L, R, A, OR Q.'
[7]     →L0
[8]    L1: LISTDIRECTORY
[9]     →L0
[10]   L2: READREC
[11]    →L0
[12]   L3: ADDREC
[13]    →L0
[14]   L4: ⎕FUNTIE 1111 2222
[15]   ∇
```

branching to more than one line

Notice that line [5] uses a new branching technique. The technique, which uses compression, allows the function to branch to different lines, depending on the response to the question on line [3]. If you enter *A* in response to the prompt, the function executes:

```
      'LRAQ'='A'
0 0 1 0
```

The function then uses this vector as a compression vector in the rest of the expression:

```
      0 0 1 0/L1,L2,L3,L4
```

The 1 in the compression vector matches the label *L*3, and the system goes to line [12] and runs the *ADDREC* function. The function then returns to line [3] and asks you to select another function — *LIST*, *READ*, *ADD*, or *QUIT*. If you answer *L*, the system goes to line [8], and so on. The function always returns to line [3] until you enter *Q* for *QUIT*.

There is one other new technique in this function. Line [6] reminds you that you must enter *L*, *R*, *A*, or *Q*. Since the test (line [5]) checks only for these four letters, any other response (like *X*, *Z*, or *G*) causes the system to go on to line [6], because you did not tell it to go to *L1*, *L2*, *L3*, or *L4*. Line [6] prints a message listing the valid responses and branches back to line [3].

setting up the system

TIEFILES, the first part of the recipes system, is a simple function. It only ties two files. You want to use two files, *RECIPE* and *DIRECTRY*, so you can store the directory separately from the recipes.

```
      ∇ TIEFILES
[1]    ⍝ TIES FILES FOR RECIPE SYSTEM
[2]     'RECIPES' ⎕FTIE 1111
[3]     'DIRECTRY' ⎕FTIE 2222
[4]   ∇
```

Before you try these examples, erase the old *RECIPES* file:

```
      'RECIPES' ⎕FTIE 3371
      'RECIPES' ⎕FERASE 3371
```

Now create a new set of files for the new system:

```
      'RECIPES' ⎕FCREATE 1
      'DIRECTRY' ⎕FCREATE 2
```

(Notice that the directory file name is spelled *DIRECTRY*. This keeps the name within the eight-character limit of some APL★PLUS systems.) Since *TIEFILES* ties these files for you, untie them now:

```
      ⎕FUNTIE ⎕FNUMS
```

Next, create *LISTDIRECTORY*. Enter:

```
      ∇ LISTDIRECTORY;NUMS;DIR
[1]    ⍝  PRINTS RECIPES DIRECTORY
[2]     DIR←⎕FREAD 2222 1
[3]     NUMS←⍳1↑⍴DIR
[4]     'Q<  >I5,55A1' ⎕FMT (NUMS;DIR)
[5]   ∇
```

The 1↑ρ*DIR* expression on line [3] determines the number of rows in the directory by taking the first number of the shape of *DIR*. (Remember that rho (ρ) returns the number or rows, then columns, in a matrix.) The iota (ι) creates a vector of consecutive numbers using the number of rows. The branch function (←) assigns that vector to the variable *NUMS*.

Line [4] uses ⎕*FMT* to format the numbers in *NUMS* next to the directory entries in *DIR*. The *Q*< > formatting modifier places two spaces between the numbers and the entries.

It is important to have a separate file for the directory — that way the recipe numbers start with 1; that is, they start in component 1 of the *RECIPES* file.

READREC, the function that reads the recipe, is next. This function assumes that you have listed the directory and know the number of the recipe you want.

```
      ∇ READREC;RECNO
[1]    ⍝ DISPLAYS A RECIPE BY NUMBER
[2]     'READ WHICH RECIPE (BY NUMBER):'
[3]     RECNO←⎕
[4]     ⎕FREAD 1111,RECNO
[5]   ∇
```

READREC uses quad input to get the recipe number. It stores the number in the variable *RECNO*. Finally, *READREC* uses *RECNO* as the second element — the component number — in the argument to ⎕*FREAD*.

ADDREC is similar to *FILERECIPE*, except that it uses 55 columns for the recipe lines instead of 60. Enter the function as shown in the next example.

```
      ∇ ADDREC;RECIPE;NL;DIR;NEWENTRY
[1]    ⍝ COLLECTS AND FILES RECIPE
[2]     'ENTER YOUR RECIPE LINE BY LINE'
[3]     '(USE ¨END¨ TO STOP):'
[4]     RECIPE←55↑⍞
[5]     RECIPE← 1 55 ρRECIPE
[6]    L1:'NEXT LINE:'
[7]     NL←55↑⍞
[8]     →(∧/'END'=3↑NL)ρL2
[9]     RECIPE←RECIPE,[1]NL
[10]    →L1
```

```
[11] L2:RECIPE ⎕FAPPEND 1111
[12]  DIR←⎕FREAD 2222 1
[13]  NEWENTRY←RECIPE[1;]
[14]  DIR←DIR,[1]NEWENTRY
[15]  DIR ⎕FREPLACE 2222 1
[16]  'NEW RECIPE FILED.'
[17] ∇
```

Just as in *FILERECIPE*, you continue to add lines until you enter *END*. Then the system stores the recipe in the file.

Lines [12] through [15] keep the directory up-to-date by reading in the old directory, getting the new title, and adding it to the bottom of the directory matrix. The expression *NEWENTRY←RECIPE*[1;] on line [13] takes the first row of the recipe — the title. Line [15] returns the new, updated directory to the file.

using an empty matrix to start the system

To start this system, you have to put something in the *DIRECTRY* file so that *ADDREC* finds something when it reads the first component (line [12] of *ADDREC*). Otherwise, APL★PLUS displays *FILE INDEX ERROR*. Because you do not really want anything in the directory yet, you fool the system by putting an empty matrix with 55 blank columns and *no* rows in the directory file. The expression

```
      0 55ρ' '
```

creates this empty matrix. Enter:

```
      TIEFILES
      BLANKROW←0 55ρ' '
      BLANKROW ⎕FAPPEND 2222
      ⎕FUNTIE ⎕FNUMS
```

using the system

Now you are ready to use the system. The following example shows how to add a recipe, list the directory, read a recipe, and stop the system.

```
      RECIPES
LIST, READ, ADD, OR QUIT?
(L, R, A, OR Q):
```

(entering a recipe)

```
A
ENTER YOUR RECIPE LINE BY LINE
(USE "END" TO STOP):
GOAT STEW
NEXT LINE:
FIND LARGE GOAT.
NEXT LINE:
PUT IN LARGE KETTLE WITH CAMP AX.
NEXT LINE:
BOIL 12 HOURS.
NEXT LINE:
THROW AWAY GOAT AND EAT AX.
NEXT LINE:
END
NEW RECIPE FILED.
LIST, READ, ADD, OR QUIT?
(L, R, A, OR Q):
```

(listing the directory)

```
L
 1   GOAT STEW
LIST, READ, ADD, OR QUIT?
(L, R, A, OR Q):
```

(reading a recipe)

```
R
READ WHICH RECIPE (BY NUMBER):
⎕:
      1
GOAT STEW
FIND LARGE GOAT.
PUT IN LARGE KETTLE WITH CAMP AX.
BOIL 12 HOURS.
THROW AWAY GOAT AND EAT AX.
LIST, READ, ADD, OR QUIT?
(L, R, A, OR Q):
```

(ending the system)

```
Q
```

Summary

In this chapter, you learned how to use the file system functions shown in Table 10-1. Remember, many of these commands assume that you have a file tied. If a *FILE TIE ERROR* occurs when you use one of the file functions, you probably do not have a file tied.

Table 10-1. File System Functions

File Function Syntax	Description
data ⎕*FAPPEND* *tie number*	Appends data to the end of a file.
'*name*' ⎕*FCREATE* *tie number*	Creates (and ties) a file.
'*name*' ⎕*FERASE* *tie number*	Erases a file.
⎕*FLIB* '' or ⎕*FLIB* *identifier*	Lists the files in a library.
⎕*FNAMES*	Lists the names of files currently tied.
⎕*FNUMS*	Lists the tie numbers of files currently tied.
⎕*FREAD* *tie number* *comp number*	Reads the data in one component of a file.
data ⎕*FREPLACE* *tie number* *comp number*	Replaces data in a component with new data.
⎕*FSIZE* *tie number*	Shows the size of a file.
'*name*' ⎕*FTIE* *tie number*	Ties a file.
⎕*FUNTIE* *tie number(s)*	Unties a file or files

Exercises

A. 1. Create a file named *EXERCISE*.

2. Create a variable called *MSG* containing the message:

   FILE FOR EXERCISES IN CHAPTER 10.

   This is just an informative message to identify the file. Append it to the file *EXERCISE*.

3. Create a variable called *DATA* — a 3-by-3 matrix containing the numbers 110, 150, 117, 35, 65, 79, 84, 114, 97. Append it to the end of *EXERCISE*.

4. Create a variable called *DATA2* — a vector containing the numbers 11.52, 3.72, 6.15. Append it to the end of *EXERCISE*.

5. Read in the data in component 2 and assign it to a variable named *ITEMS*. (Remember, you can do this with a single APL statement.)

6. Read in the data from component 3 and assign it to a variable named *PRICES*.

7. Write an APL expression to find the third price in *PRICES*. (Hint: Use indexing.)

8. Write an APL expression to find the third row of *ITEMS*.

9. You sell nuts, bolts, and wrenches in your store. Row 1 of *ITEMS* shows how many nuts you sold in each of the past three weeks, and rows 2 and 3 show the same thing for bolts and wrenches. Write an APL expression that calculates the *total* number of nuts, bolts, and wrenches you sold in the past three weeks. (Hint: Use plus reduction.)

10. Write an APL expression that calculates the total number of *wrenches only* sold during the three weeks. That is, find the row total for row 3 of *ITEMS* only.

11. *PRICES* contains the prices you charge for nuts, bolts, and wrenches. Write an APL expression that calculates how much you charged for all the wrenches you sold during the past three weeks.

B. 1. Untie the *EXERCISE* file.

2. Define a function called *TIE* that ties *EXERCISE* to the number 277.

3. Execute the function *TIE*. Enter ⎕*FNAMES* and then ⎕*FNUMS* and check to see whether the function worked.

4. Define a function called *READ* that displays the number of sales for the past three weeks from the file *EXERCISE*. That is, it displays the entire matrix stored in component 2. (Remember that the file is tied to 277.) Run the function.

5. Write a function called *TOTSALES* that reads in both components 2 and 3. It then totals the unit sales given in component 2 (finds total sales for each item for three weeks). Finally, it multiplies the totals by the prices to find the total sales. Run the function. (Hints: In the function, assign the unit sales to the variable *ITEMS*, and the prices to the variable *PRICES*. The rest of the function is similar to Exercise A.11.)

C. File system functions work with variables much as they work with numbers. For example, you can enter:

```
      TIENO←1
      'TEST' ⎕FTIE TIENO
      ⎕FNUMS
1
```

(If you did not untie the *TEST* file before entering the preceding example, you may have gotten an error message.)

```
      1000 ⎕FAPPEND TIENO
      ⎕FREAD TIENO,1
1000
```

(The comma combines the variable *TIENO* with the 1.)

Since file functions can use variables, you can assign component numbers to meaningful names and then use the names to read data. Enter:

```
      M←'FILE FOR TESTING EXAMPLES.'
      M ⎕FREPLACE TIENO,1
      MESSAGE←1
      ⎕FREAD TIENO, MESSAGE
FILE FOR TESTING EXAMPLES.
      ⎕FREAD 1 1
FILE FOR TESTING EXAMPLES.
```

You can put all of this in a function. First, erase *READ*:

```
      )ERASE READ
```

Then define a new *READ* function:

```
      ∇READ DATA
[1]   ⎕FREAD TIENO,DATA
[2]   ∇
```

Now enter:

```
       READ MESSAGE
FILE FOR TESTING EXAMPLES.
```

1. Enter *)CLEAR* to clear the workspace. Define a *READ* function to read in data from *EXERCISE*. The function should be able to do the following:

```
      READ ITEMS
110 150 117
 35  65  79
 84 114  97
      READ PRICES
11.52 3.72 6.15
```

(Hints: This function is similar to the *READ* function you defined above, but it uses the tie number for *EXERCISE*. Remember to assign the correct component numbers to the variables *PRICES* and *ITEMS* before trying to run *READ*.)

2. Write a function *TOTAL* that reads in a matrix and totals the rows in the matrix. You should be able to read in your unit sales data from *EXERCISE* and total it like so:

```
      TOTAL ITEMS
377 179 295
```

3. Create a function called *TOTSALE* that calculates how much you made on a particular item in each of the three weeks. That is, the result should be three numbers — each one is the unit sales for the week multiplied by the price of the item. Your function should display the following results:

```
      TOTSALE NUTS
1267.2 1728 1347.84
      TOTSALE BOLTS
130.2 241.8 293.88
```

The function should:

- read in the unit sales data
- read in the prices
- find the correct row in the unit sales matrix *ITEMS*
- find the correct price in the variable *PRICES*
- multiply the price by the correct row.

(Hint: To find the correct row, use a variable with indexing:

```
      D←3 5⍴⍳15
      D
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15

      X←2
      D[X;]
6 7 8 9 10
```

This also works for vectors:

```
      E←⍳5
      E[X]
2
```

To make your function work, you have to assign the row numbers of items to variables:

```
      NUTS←1
      BOLTS←2
      WRENCHES←3
```

Now you can use these variables when you index.)

D. 1. Create a file called *SALES*.

2. Create the following variables.

```
      SNOS←101 102 103
      MON←1 2 3 4
      SALES←3 4⍴107 62 97 108 80
72 95 67 88 92 101 81
```

Append these variables to the *SALES* file (first *SNOS*, then *MON*, then *SALES*).

SNOS are salesman numbers, and *MON* are the numbers for four months. *SNOS* corresponds to the rows of *SALES*; *MON* corresponds to the columns.

3. Untie the *SALES* file.

4. Write a function called *GETDATA* that:

- Ties *SALES* to the number 1076.

- Reads in each of the three components in *SALES* and assigns them to the variables *SNOS*, *MON*, and *SALES*. (This is done so that the data is in the workspace where you can work on it — there is no need for every function to read components from the file.)

Run the function.

5. Create the following function *SALESMAN*, which assigns the number you specify to a variable *S*.

```
          ∇SALESMAN NUM
[1]        S←NUM
[2]        ∇
```

6. Create a function *MONTH* that is similar to the *SALESMAN* function. *MONTH* assigns the number you specify to a variable *M*.

7. Create a function *PRINT* that uses *M* and *S* to find the desired element of *SALES*. The functions should behave like this:

```
          SALESMAN 101
          MONTH 4
          PRINT
108
```

The function *PRINT* chooses the row for salesman 101 (row 1) and the column for month 4 (column 4).

(Hint: Use the selection techniques you learned in the Selecting and Analyzing Data chapter. For example, to get the result shown above, you could use:

```
          (MON=4)/(SNOS=101)/[1]SALES
108
```

Remember that 101 and 4 were stored in variables by the functions *SALESMAN* and *MONTH*, though.)

Notice that you have created a complete system. A complete run using this system might go like this:

```
          GETDATA
          SALESMAN 103
          MONTH 1
          PRINT
88
          )OFF
FILES UNTIED
```

If someone created this system for you, you could use it without ever knowing what a file is.

E. Design and create a system for collecting information on your household goods. The system should contain three main functions, *INVSTART*, *ENTER*, and *LIST*, and two *subfunctions*, *GETNAME* and *GETDETAILS*. (A subfunction is a function called by another function.) *INVSTART* ties the inventory file. *ENTER* collects data on your household inventory and files it. *LIST* reads in the data and displays it. The *LIST* function appears below.

```
      ∇ LIST;DIR;NUMS;NUMBER;NAMES;NAM;DET
[1]     ⍝ MANAGES HOUSEHOLD INVENTORY SYSTEM
[2]     L1: 'DIRECTORY, ITEMS, OR QUIT? (D,I,Q):'
[3]      →('DIQ'=1↑))/DI,ITEMS,EXIT
[4]      'PLEASE ENTER D, I, OR Q'
[5]      →L1
[6]     DI: DIR←⎕FREAD 1068 2
[7]      NUMS←⍳1↑⍴DIR
[8]      'Q<  >I5,20A1' ⎕FMT (NUMS;DIR)
[9]      →L1
[10]    ITEMS: 'LIST WHICH ITEM (BY NUMBER)?'
[11]     NUMBER←⎕
[12]     NAMES←⎕FREAD 1068 2
[13]     DETAILS←⎕FREAD 1068 3
[14]     NAM←NAMES[NUMBER;]
[15]     DET←DETAILS[NUMBER;]
[16]     NAM,DET
[17]     →L1
[18]    EXIT:
      ∇
```

LIST reads and lists either the directory or the detailed description of an item, similar to *RECIPES*. The only difference is that individual items are merely single lines (consisting of two parts — the item name and the item details) instead of an entire matrix.

Since the items you want in are single lines, you can store them in two matrices (one for names, one for details). These matrices are located in components 2 and 3 of the file *INVENTOR*. The function *LIST* uses the expressions *NAMES*[*NUMBER*;] and *DETAILS*[*NUMBER*;] to select the rows (item name and details) you want. Then it prints the two parts with the expression *NAM*,*DET*.

Notice that *LIST* uses the *NAMES* matrix for the directory. This is why you store the names and details in different components.

ENTER calls two subfunctions, *GETNAME* and *GETDETAIL*. This function appears below.

```
      ∇ ENTER
[1]     GETNAME
[2]     GETDETAILS
[3]     'NEW ITEM FILED.'
      ∇
```

The functions left for you to create are *INVSTART*, *GETNAME*, and *GETDETAILS*. *GETNAME* and *GETDETAILS* are similar to the first few lines of *FILERECIPE* and *ADDREC*. They read in the contents of the file (names and details), gather a single line, add it to the bottom of the matrix, and replace the matrix in the file. The names column should have a field width of 20 characters (20↑⍞) and the details column should have a field width of 35 characters. (Actually, you can use larger field widths, but your solution uses 20 and 35.)

A sample run of *LIST* might look like this:

```
      INVSTART
      ENTER
NAME OF ITEM:
19 IN. COLOR TV
SERIAL NUMBER, DATE PURCHASED, PRICE:
R8254A    05/11/91    $421.63
NEW ITEM FILED.
```

```
      LIST
DIRECTORY, ITEMS, OR QUIT?  (D,I,Q):
D
1  19 IN. COLOR TV
DIRECTORY, ITEMS, OR QUIT?  (D,I,Q):
I
LIST WHICH ITEM (BY NUMBER)?
⎕:
      1
19 IN. COLOR TV    R8254A    05/11/91
$421.63
DIRECTORY, ITEMS, OR QUIT?  (D,I,Q):
Q
```

Notice that, since the details are stored in one line, it is up to you to space the three items correctly. You left four spaces between the detail items, which works well.

Before running your system, be sure to start it properly:

```
      'INVENTOR' ⎕FCREATE 5
      M←'FILE FOR STORING INVENTORY
DATA.  USE <INVSTART>, <ENTER>, AND
<LIST> WITH FILE.'
      M ⎕FAPPEND 5
      (0 20⍴' ') ⎕FAPPEND 5
      (0 35⍴' ') ⎕FAPPEND 5
```

(These last two steps are necessary because *ENTER* will look for something in the file.)

```
      ⎕FUNTIE 5
```

There are lots of ways you can enhance this system. For example, you can add items to the details list, such as place purchased, location of inventory, and so on. You can also store each detail in a separate component and have *LIST* ask you which details you want to see.

F. Design a system to collect and format names and figures for a household budget. The function *BUDGET* collects and formats the data. The function *SHOWBUDGET* lists the data that *BUDGET* collected.

A sample run using *BUDGET* might look like this:

```
        BUDGET
ENTER ITEMS ("END" TO STOP):
FOOD
NEXT ITEM:
RENT
NEXT ITEM:
GASOLINE
NEXT ITEM:
ENTERTAINMENT
NEXT ITEM:
END
ENTER BUDGET FIGURES FOR 4 ITEMS:
⎕:
        250 431.72 85 125
BUDGET IS:

FOOD                    $250.00     28.0%
RENT                    $431.72     48.4%
GASOLINE                 $85.00      9.5%
ENTERTAINMENT           $125.00     14.0%
-----------------------------------------

TOTAL                   $891.72    100.0%

REMEMBER TO )SAVE THE FINAL VERSION.
```

You can define as many different budgets as you like by running *BUDGET* again. The function will always tell you what the total budget amount is and what percentage each item is of the total. You use *BUDGET* until you design a budget that you want to save, then you enter *)SAVE*. Once you save a budget, run *SHOWBUDGET*, which uses the global variables created by *BUDGET*. *SHOWBUDGET* prints the same report as *BUDGET*, except that it omits the *BUDGET IS* and *REMEMBER* lines.

The APL expressions for the *SHOWBUDGET* function appear below:

```
     ∇ SHOWBUDGET
[1]   ⍝ DISPLAYS BUDGET.
[2]    ' '
[3]    FSB ⎕FMT (INAMES;N;PERCENT)
[4]    44⍴'¯'
[5]    FSB ⎕FMT (T;+/N;100)
     ∇
```

SHOWBUDGET uses the same format string (*FSB*) that *BUDGET* does, and it uses *FSB* for the body of the report as well as the total line. *SHOWBUDGET* produces the underline with the expression `44⍴'¯'`; and *T* is the word *TOTAL*, produced by the following sequence:

```
        T←20↑'TOTAL'
        T←1 20⍴T
```

The variable *N* contains the budget numbers, and the expression `+/N` calculates the total budget figure. To find the total percentage figure — use 100 instead of adding up the percentages. (The total may be 99.9% or so, due to rounding.)

BUDGET is similar to the other data-gathering functions you have learned. Two new tricks are included in it, though, which are explained below.

To produce the line

```
        ENTER BUDGET FIGURES FOR 4 ITEMS:
```

use the APL expression:

```
'ENTER BUDGET FIGURES FOR ',(⍕1↑⍴INAMES),
' ITEMS:'
```

The expression in parentheses takes the first number of the shape of *INAMES* (item names), just like you did to produce the directory in Exercise B. Then it converts the number (the number of rows in *INAMES*) into character data.

To calculate the percentage figures, total the budget figures (*TOT*←+/*N*) and then divide each figure by the total and multiply by 100:

```
PERCENT←100×N÷TOT
```

A

Solutions to Exercises

Chapter 1

A. 1. `4 5 6`

2. `3`

3. `¯2`

4. `6 12`

5. `6 12`

This is the same answer as for Exercise 4, since in both exercises the system multiplies 3 by both 2 and 4.

6. *SYNTAX ERROR*

The plus function (`+`) expects data on both sides. That is, plus is a dyadic function.

7. *SYNTAX ERROR*

Since the high minus symbol (`¯`) is part of the number 2 (`¯2` = negative two), this expression has no meaning. Use the middle minus symbol (`-`) to subtract; for example, `3-2`.

8. `5 2 3`

9. *DOMAIN ERROR*

Division by zero is not defined in arithmetic. Thus, zero is outside of the domain of the division function.

10. 40

Remember to evaluate right to left; that is, 2+6=8, 5×8=40.

11. *LENGTH ERROR*

The two vectors are not equal in length; the system cannot add them.

12. 16

The parentheses change the order of execution.

13. *SYNTAX ERROR*

In regular math, parentheses can show multiplication; in APL, the times sign (×) shows multiplication.

14. 4 2

15. 8

16. 1

B. 1. 4

2. 3 6 9

3.
```
2 3 4
5 6 7
```

4. 2 8 14

5.
```
0 2  4
6 8 10
```

6. *RANK ERROR*

You cannot add vectors to matrices.

7.
```
4 5 6
7 8 9
```

8. 8 6 4

9. *LENGTH ERROR*

The vector on the left contains only two elements, but *B* contains three.

10. `2 8 14`

C. 1. `3`

2. `4`

3. `7`

Since APL evaluates right to left, it links the two vectors together and then takes the shape of the new, longer vector.

4. `0 0 1 5`

The system catenates a scalar to the end of a vector.

5. `4`

The new vector has four elements.

6. `1 2 4 9`

Since the new vector has four elements and *V2* also has four elements, the system can add the two vectors.

7. `4 4 4`

You use the dyadic reshape function (ρ) to create a vector. Instead of rows and columns, you get one string, with as many elements as the number on the left specifies.

8.
```
2 1 2
1 2 1
2 1 2
```

9.
```
0
0
```

Here, the reshape function (ρ) uses the contents of *V*3 (2 1) to reshape *V* into a matrix with two rows and one column. Since the 1 is not required (*V*1 contains 0 0 1), APL omits it.

10.
```
5 0 0 1
```

The system catenates a 5 to the front of a vector.

Questions 4, 6, 8, and 9 point out some additional properties of the reshape (ρ) and catenate (,) functions rather than test you on your previous knowledge. Try some problems of your own with these functions and try to interpret the results.

D. 1.

```
      .20 .30 .50×12 6 2
2.4 1.8 1
```

or

```
      12 6 2×.20 .30 .50
```

E.

```
      600 700 1000 950×.05 .0525 .055 .06
30 36.75 55 57
```

F.

```
      100×142 167 161 128÷175
81.14285714 95.42857143 92 73.142855714
```

G.

```
      2116+(7216-6000)×.25
2420
```

You can also use `2116+.25×7216-6000`. This second expression works because the system first takes the difference (`7216-6000`), then performs the multiplication (`.25×1216`), and then the addition (`2116+304`). Remember, APL evaluates expressions from right to left.

H.

```
      SALES←2 3⍴12 9 3 11 12 6
      PRICES←2 3⍴3.25 1.79 2.55 3.45 1.85 2.75
      SALES×PRICES
39    16.11  7.65
37.95 22.2   16.5
```

I.

```
      GRADES←90 82 79 92
      OLDGRADES←GRADES
      GRADES←93 87 80 90 78 85
      GRADES
93 87 80 90 78 85
      OLDGRADES
90 82 79 92
```

J.

```
      INVCOST←QUANT×COST
      INVCOST
472.5 773.67 704.48 155.25
```

Chapter 2

A. 1.

```
      ARROWS←'←→↑↓'
      ARROWS
←→↑↓
```

2.

```
      CHARS ← '¨¯<≤=≥>≠∨∧→⋆○⍳↓↑~⍴∊⍵?⍺⌈
⌊_∇∆∘'''⎕()⊂⊃∩∪⊥⊤|;:\'
```

Of course, the order will vary depending on how you type in the symbols. You may have trouble with the single quote. You must have an even number of quotes in a line, so enter an extra quote if you run into a problem. Then reassign the variable, and remember to double the quote.

3.

```
      NUMBERS←3 5⍴'ONE  TWO  THREE'
      NUMBERS
ONE
TWO
THREE
```

4.

```
      ALPH←5 1ρ'ABCDE'
      ALPH
A
B
C
D
E
```

5. The shape of *CV* is 4, since the expression in Question 5 contains four spaces. If you entered a different number of spaces, the shape will be different.

6. The shape is 8. Since APL translates the double quotes back to a single quote, the double quotes count as only one element.

7.

```
      A[5 2 3]
TAN
```

8.

```
      B,C
BLUEBELL
      D,C
JINGLEBELL
      C,E
BELLRINGER
```

9. a.

```
      ALPH[20]
T
```

b.

```
      SYMB[4]
⋆
```

c.

```
      ALPH,SYMB
ABCDEFGHIJKLMNOPQRSTUVWXYZ∇∆|⋆∘
```

d.

```
      ρALPH,SYMB
31
```

e.

```
      ALPH[⍳12]
ABCDEFGHIJKL
```

or

```
      ALPH[1 2 3 4 5 6 7 8 9 10 11 12]
```

or even

```
      12ρALPH
```

```
f.         ALPH[?26 26 26]
```

or

```
              ALPH[?3ρ26]
```

Try the expression a few times to see the result.

```
10.       10ρ'*'
    **********
```

You asked APL to reshape the single star into a character vector with ten stars.

```
11.       M←'THE NUMBER IS '
          M,⍕20
    THE NUMBER IS 20
```

```
12.       P1←'YOU HAVE SPENT '
          P2←' DOLLARS.'
          P1,(⍕10),P2
    YOU HAVE SPENT 10 DOLLARS.
```

You need parentheses around the numeric portion of the statement (⍕10). This prevents the catenate function (,) from joining the character data in *P2* to the number 10 before APL converts *P2* to character data. If you forgot the parentheses, APL probably displayed *DOMAIN ERROR*, since the catenate function is not defined for use with mixed character and numeric data.

B. 1. 4 4 7 6

2. 3 1 4 4

3. 4 2 2 8

4. 2 1 2 2

5. 4 1 7 8

6.
```
4 1 7 8
```

For this function, the data can appear on either side of the function symbol.

7.
```
3 1 2 6
```

8.
```
4 3 3
9 7 3
```

9.
```
3 3 5
6 8 3
```

10.
```
4 2 3
4 4 1
```

11.
```
1 3 4
4 4 1
```

12.
```
4 2 3
9 7 1
```

You can use the maximum function to compare *C* to itself, but the result is *C* again.

13.
```
4 3 5
9 8 1
```

14.
```
1 2 3
6 7 1
```

C. 1.
```
17
```

2.
```
15
```

3.
```
9 17
```

4.
```
9 15
```

5.
```
8
```

6.
```
1
```

7.
```
4 9
```

8. `1 1`

9. `13 9 4`

10. `7 11 6`

11. `26`

12. `24`

13. `9`

14. `1`

D. 1.

```
      TRANSACT[3]
17
```

2.

```
      +/TRANSACT
154.22
```

3.

```
      +\TRANSACT
153.17 133.17 150.17 114.42 141.97 154.22
```

4.

```
      TRANSACT[1]-TRANSACT[6]
140.92
```

If you solved this problem, congratulations! It is different from the material covered in the chapter.

5.

```
      (+/TRANSACT)×.05
7.711
```

or

```
      .05×+/TRANSACT
```

The first expression is closer to regular arithmetic, but the second one saves typing two parentheses by using the APL right-to-left execution.

E. 1.

```
      ⍴DEPOSITS
8
```

2.

```
      (+/DEPOSITS)-+/WITHDRAWALS
697.72
```

The parentheses on the left ensure that the system subtracts total withdrawals from total deposits. Without parentheses on the left, the system subtracts the total withdrawals from each deposit, then adds all three negative numbers. Try it.

3.

```
      DEPOSITS←DEPOSITS,450
      DEPOSITS
120 25 115 375 800.12 22 75 82.75 450
```

4.

```
      ⌈/DEPOSITS
800.12
      ⌊/DEPOSITS
22
```

5.

```
      ⌈/WITHDRAWALS
372.12
      ⌊/WITHDRAWALS
15
```

6.

```
      (⌈/DEPOSITS)-⌈/WITHDRAWALS
428
```

Remember the parentheses!

7.

```
      DEPOSITS[1 3 5]
120 115 800.12
```

8.

```
      WITHDRAWALS[⍳5]
35 100 220 175 372.12
```

or

```
      WITHDRAWALS[1 2 3 4 5]
```

or

```
      5⍴WITHDRAWALS
```

F. 1.

```
      Z
HOW
CAN
THAT
BE?
```

2.

```
      Z[1;1 2 3]
HOW
```

3.

```
      Z[3;2 3 4]
HAT
```

4.

```
      Z[4;3]
?
```

5.

```
      Z[4;3 3 3]
???
```

or

```
      3ρZ[4;3]
```

6.

```
      Z[?4;1]
```

Enter the expression to see what happens.

7.

```
      Z[2;?4]
```

Remember that there is a blank in this row, so the result may be "invisible" if the system chooses the random number 4.

Chapter 3

A. 1.

```
      +/'E'='TENNESSEE'
4
```

2. a.

```
      +/TRANSACT<0
3
```

TRANSACT<0 gives a result of 1s and 0s; the +/ adds up the 1s, which correspond to negative numbers (that is, withdrawals).

b.

```
              +/TRANSACT≥0
    4
```

3.

```
          +/'  '=S
9
```

4.

```
          +/SALES≤375
5
```

B. 1.

```
          ∧/ACCOUNTS≥0
0
```

The answer is no — some accounts must have negative balances.

2.

```
          +/ACCOUNTS<0
3
```

3.

```
          +/ACCOUNTS≥0
7
```

4.

```
          ∨/ACCOUNTS>200
1
```

The answer is yes — there must at least one account greater than $200. Enter:

```
          +/ACCOUNTS>200
1
```

There is only one.

5.

```
          ∨/(ACCOUNTS>120)∧ACCOUNTS<175
1
```

Yes, there are some accounts between $120 and $175.

```
          +/(ACCOUNTS>120)∧ACCOUNTS<175
2
```

In fact, there are two accounts between $120 and $175.

6.

```
      +/(ACCOUNTS<0)/ACCOUNTS
¯97
```

(ACCOUNTS<0)/ uses the compression function to pick out the negative balances, then the +/ adds them.

7.

```
      +/(ACCOUNTS≥0)/ACCOUNTS
633
```

C. 1.

```
      (PRICES>15)/PRICES
30.29 18.95 19.95
```

Remember the parentheses around the expression on the left. Without the parentheses, APL displays either *DOMAIN ERROR* or *LENGTH ERROR*.

2.

```
      (PRICES≤3.07)/PRICES
1.79 3.07 2.15
```

3.

```
      (PRICES=17.75)/PRICES
```

This expression gives no result, because none of the prices equals $17.75.

4.

```
      ((PRICES>10)∧PRICES<20)/PRICES
10.77 18.95 19.95
```

You may have had trouble with all of the parentheses in this exercise. Remember that you must always have an *even* number of parentheses (just like quotes). If you omit any parentheses, the system displays an error message.

5.

```
      ((PRICES<2)∨PRICES>30)/PRICES
1.79 30.29
```

D. 1.

```
 3
 6
 9
```

2.

```
OE
OE
OE
```

3. *MOE*

4. 4 5 6

5. 6

   The rightmost part (0 1 0⌿*A*) chooses the middle row of *A*, then the 0 0 1 part chooses the final element of that row.

6. *LENGTH ERROR*

   A has only three elements.

E. 1.

```
      (CUSTNO=3)⌿CORDERS
72 35 99
```

The ⌿ is necessary because you want a *row* of orders. Remember, each row corresponds to one customer.

2.

```
      (CUSTNO=4)⌿CADDRESS
27 MEMORIAL DR.
```

Again, you need a row of the address matrix.

3.

```
      (MONTHS=11)/CORDERS
412
 84
 35
 78
```

This time you need a *column*, because you are searching by month rather than by customer.

4.

```
     (CUSTNO=1)⌿(MONTHS=12)/CORDERS
811
```

or

```
     (MONTHS=12)/(CUSTNO=1)⌿CORDERS
```

Either expression works, but remember to use ⌿ with *CUSTNO*, because the length of *CUSTNO* matches the number of *rows* in *CORDERS*. If you enter (*CUSTNO*=1)/*CORDER*, the system displays *LENGTH*

ERROR, because you are trying to compress three columns using four 0s and 1s.

Chapter 6

Note: You may have different names for the function arguments than the ones shown here.

A. The answer shows all of the steps in one continuous display.

```
      ∇ NUMBERS
[6]   [2]
[2]   TOO
[3]   [4]
[4]   FORE
[5]   [3⎕0]
[3]   THREE
[3]   TREE
[4]   [1.5]
[1.5]   TWO
[1.6]   [~5]
[5]   [⎕]
    ∇ NUMBERS
[1]   ONE
[1.5]   TWO
[2]   TOO
[3]   TREE
[4]   FORE
    ∇
[5]   [0]
[0]   COUNT
[5]   [⎕]
    ∇ COUNT
[1]   ONE
[1.5]   TWO
[2]   TOO
[3]   TREE
[4]   FORE
    ∇
[5]   ∇
```

Remember to type a del symbol (∇) to leave function definition mode.

B. 1. `R←A ADD B`

2. `R←A+B`

3.

```
      ∇R←A ADD B
[1]   R←A+B
[2]   ∇
      A←4 ADD 2
      A
6
```

4. `RUNTOT DATA`

5. `+\DATA`

6.

```
      ∇RUNTOT DATA
[1]   +\DATA
[2]   ∇
      RUNTOT 10 20 30 40
10 30 60 100
```

C. 1.

```
      ∇HI
[1]   'HELLO.  MY NAME''S APL.'
[2]   'WHAT''S YOURS?'
[3]   ∇
```

Remember to double the quotes.

2.

```
      ∇HALF NUM
[1]   NUM×.5
[2]   ∇
      HALF 3.14159
1.570795
```

3.

```
      ∇HALF
[2]   [0]
[0]   R←HALF NUM
[2]   [1]
[1]   R←NUM×.5
[2]   [⎕]
      ∇R←HALF NUM
[1]   R←NUM×.5
    ∇
[2]   ∇
```

You *must* remember to assign a result both in the header *and* in the body of the function. If you do not, the system displays a *VALUE ERROR* in Exercise 4.

4.

```
      10+HALF 3.14159
11.5070795
```

5.

```
      MAT←3 2⍴⍳6
      ∇ROWTOT MAT
[1]   +/MAT
[2]   ∇
      ROWTOT MAT
3 7 11
```

6.

```
      ∇COLTOT MAT
[1]   +⌿MAT
[2]   ∇
      COLTOT MAT
9 12
```

7.

```
      ROWTOT MAT2
50 66 82
     COLTOT MAT2
45 48 51 54
```

8.

```
      ∇RPT DATA
[1]   DATA
[2]   '-----------'
[3]   COLTOT DATA
[4]   ∇
      RPT MAT2
 11 12 13 14
 15 16 17 18
 10 20 21 22
-----------
45 48 51 54
```

D. 1.

```
          ∇REPEAT X
[1]       X,' ',X,' ',X,' ',X,' ',X
[2]       ∇
```

REPEAT takes the character data you enter (such as '*ANN*'), catenates it to a single blank (' '), takes it again, catenates it to another blank, and so on.

```
          REPEAT 'ANN'
ANN ANN ANN ANN ANN
```

2.

```
       ∇SQUARE X
[1]       'THE NUMBER SQUARED IS ',⍕X×X
[2]       ∇
           SQUARE 247
THE NUMBER SQUARED IS 61009
```

3.

```
          ∇A CHOOSE B
[1]       A[B]
[2]       ∇
8 7 12 73 55 CHOOSE 3
12
101 25 14 47 CHOOSE 2
25
NOS←8 76 43 21 23 6
           NOS CHOOSE 4
21
```

4.

```
          ∇X OF Y
[1]       XρY
[2]       ∇
          4 OF '⎕'
⎕⎕⎕⎕
```

5.

```
          ∇HIST D
[1]       D[1] OF '⎕'
[2]       D[2] OF '⎕'
[3]       D[3] OF '⎕'
[4]       D[4] OF '⎕'
[5]       ∇
           HIST 4 1 6 7
⎕⎕⎕⎕
⎕
⎕⎕⎕⎕⎕⎕
⎕⎕⎕⎕⎕⎕
```

E. 1.

```
      ∇R←A GT B
[1]   R←(A>B)/A
[2]   ∇
      SALES GT 200
350 201 279
```

2.

```
      ∇R←A LT B
[1]   R←(A<B)/A
[2]   ∇
      SALES LT 200
125 115
```

3.

```
      ∇R←A EQ B
[1]   R←(A=B)/B
[2]   ∇
      SALES EQ 350
350
```

F.

```
      ∇DEPOSIT AMOUNT
[1]   DEPOS←DEPOS,AMOUNT
[2]   ∇

      ∇CHECK AMOUNT
[1]   WITHD←WITHD,AMOUNT
[2]   ∇

      ∇NEWBAL
[1]   (+/DEPOS)-+/WITHD
[2]   ∇
```

You may have forgotten the parentheses around +/*DEPOS* in *NEWBAL*. The function works, but you do not get the correct result. Instead, the system subtracts the total withdrawals from *every* deposit (*DEPOS*-+/*WITHD*), then adds those results (+/).

Chapter 7

You may use different variable and argument names in the function headers than those shown here. Also, remember that there are *many* ways to solve a problem in APL. The solutions shown here are simple and direct, but they are certainly not the *only* solutions.

A. 1.

```
           ∇ ASKDIE
    [1]    L1: ?6
    [2]    'TRY AGAIN?  (YES OR NO):'
    [3]    →('Y'=1↑⍞)⍴L1
    [4]    ∇
```

2.

```
           ∇ ADDER;NOS
    [1]    'ENTER NUMBERS TO ADD: '
    [2]    NOS←⎕
    [3]    'THE TOTAL IS ',(⍕+/NOS),'.'
    [4]    ∇
```

You may not have used a local variable, but it is good programming practice.

3.

```
           ∇ ASKREPEAT;THING;X
    [1]    'ENTER THING TO BE REPEATED:'
    [2]    THING←⍞
    [3]    'REPEAT HOW MANY TIMES?'
    [4]    X←⎕
    [5]    X⍴THING
    [6]    ∇
```

4.

```
           ∇ ASKHIST;NUMS
    [1]    'ENTER FOUR NUMBERS FOR CHART:'
    [2]    NUMS←⎕
    [3]    HIST NUMS
    [4]    ∇
```

Chapter 8

A. 1.

```
          'I8' ⎕FMT D
    10251   17513   18016
    11206   16867   19197
```

Remember to put quotes around the format phrases; otherwise, the system displays a *VALUE ERROR*.

2.

```
          'F10.2' ⎕FMT D
    10250.50  17512.70  18016.20
    11205.81  16867.12  19197.20
```

3.

```
      'F8.2' ⎕FMT D
10250.5017512.7018016.20
11205.8116867.1219197.20
```

There was enough room to format the numbers, but there is no space between the numbers.

4.

```
      'F18.2' ⎕FMT D
          10250.50            17512.70            18016.20
          11205.81            16867.12            19197.20
```

5.

```
      'CF18.2' ⎕FMT D
         10,250.50           17,512.70           18,016.20
         11,205.81           16,867.12           19,197.20
```

6.

```
      'CP<$>F18.2' ⎕FMT D
        $10,250.50          $17,512.70          $18,016.20
        $11,205.81          $16,867.12          $19,197.20
```

The order of the phrases *C* and *P* makes no difference.

B. 1.

```
      'F12.2,F10.1' ⎕FMT E
   18147.70      20.4
   19717.11     ¯17.3
```

Remember to separate format phrases with a comma to avoid a *FORMAT ERROR*.

2.

```
      'CP<$>F12.2,F10.1' ⎕FMT E
  $18,147.70      20.4
  $19,717.11     ¯17.3
```

3.

```
      'CP<$>F12.2,N<%>Q<%>F8.1' ⎕FMT E
  $18,147.70   20.4%
  $19,717.11  ¯17.3%
```

If you used only *N<%>*, you only got a percentage sign next to the negative number. If you used only *Q<%>*, you only got a percentage sign next to the positive number. Again, the *order* of the format phrases is not important.

4.

```
      'CP<$>F12.2,M<->N<%>Q<%>F8.1' ⎕FMT E
  $18,147.70   20.4%
  $19,717.11  -17.3%
```

5.

```
        'CP<$>F12.2,M<->N<%>P<+>Q<%> F8.1' ⎕FMT E
  $18,147.70  +20.4%
  $19,717.11  -17.3%
```

6.

```
      '8A1,CP<$>F12.2,M<->N<%>P<+>Q<%> F8.1' ⎕FMT (N;E)
REVENUES  $18,147.70  +20.4%
EXPENSES  $19,717.11  -17.3%
```

If you forgot the repetition factor for the names (8*A*1), you got stars in your result.

C. 1.

```
      'F12.2' ⎕FMT S
     6543.10     5342.55
     6231.89     7531.21
```

2.

```
      'F12.2' ⎕FMT (S;T)
     6543.10     5342.55    11885.65
     6231.89     7531.21    13763.10
```

3.

```
      'P<$>F12.2' ⎕FMT (S;T)
    $6543.10    $5342.55   $11885.65
    $6231.89    $7531.21   $13763.10
```

4.

```
      'CP<$>F12.2' ⎕FMT (S;T)
   $6,543.10   $5,342.55  $11,885.65
   $6,231.89   $7,531.21  $13,763.10
```

5.

```
      '9A1,3CP<$>F12.2' ⎕FMT (N1;S;T)
SALESREP1   $6,543.10   $5,342.55  $11,885.65
SALESREP2   $6,231.89   $7,531.21  $13,763.10
```

If you forgot the repetition factor for the numbers (a 3 before *CP*), the system displays stars for the second two numbers. This is because ⎕*FMT* tried to format the numbers with *A*, which is not allowed. If you omit the repetition factor for the names (9*A*1), the names appear with long strings of stars.

D. 1.

```
      ∇REPT DATA
[1]   TOT←+/DATA
[2]   '9A1,3CP<$>F12.2' ⎕FMT (N1;DATA;TOT)
[3]   ∇
      REPT S
SALESREP1    $6,543.10    $5,342.55   $11,885.65
SALESREP2    $6,231.89    $7,531.21   $13,763.10
```

2.

```
      ∇RPT2 DATA
[1]   '                         SALES FIGURES'
[2]   '                 WEEK1        WEEK2
TOTALS'
[3]   TOT←+/DATA
[4]   '9A1,3CP<$>F12.2' ⎕FMT (N1;DATA;TOT)
[5]   ∇
      RPT2 S
                        SALES FIGURES
                WEEK1        WEEK2        TOTALS
SALESREP1    $6,543.10    $5,342.55   $11,885.65
SALESREP2    $6,231.89    $7,531.21   $13,763.10
```

Chapter 10

A. 1.

```
      'EXERCISE' ⎕FCREATE 6
```

If you use a different tie number, substitute your tie number for the 6 in the rest of the answers to Exercise A.

2.

```
      MSG←'FILE FOR EXERCISES IN CHAPTER 10.'
      MSG ⎕FAPPEND 6
```

Remember to put the quotes around the message.

3.

```
      DATA←3 3ρ110 150 117 35 65 79 84 114 97
      DATA ⎕FAPPEND 6
```

4.

```
      DATA2←11.52 3.72 6.15
      DATA2 ⎕FAPPEND 6
```

5.

```
      ITEMS←⎕FREAD 6 2
```

6.

```
      PRICES←⎕FREAD 6 3
```

7.

```
      PRICES[3]
6.15
```

8.

```
      ITEMS[3;]
84 114 97
```

9.

```
      +/ITEMS
377 179 295
```

10.

```
      +/ITEMS[3;]
295
```

11.

```
      ITEMS[3;]×PRICES[3]
516.6 701.1 596.55
```

This expression gives you the total sales for *each* week. To find the total sales for all three weeks (the grand total):

```
      +/ITEMS[3;]×PRICES[3]
1814.25
```

or

```
      PRICES[3]×+/ITEMS[3;]
1814.25
```

B. 1.

```
      ⎕FUNTIE 6
```

2.

```
      ∇TIE
[1]   'EXERCISE' ⎕FTIE 277
[2]   ∇
```

3.

```
      TIE
      ⎕FNAMES
EXERCISE
      ⎕FNUMS
277
```

If you have more than one file currently tied, the display may be different.

4.

```
      ∇READ
[1]   ⎕FREAD 277 2
[2]   ∇
      READ
 110 150 117
  25  65  79
  84 114  97
```

5.

```
      ∇TOTSALE
[1]   ITEMS←⎕FREAD 277 2
[2]   PRICES←⎕FREAD 277 3
[3]   PRICES×+/ITEMS
[4]   ∇
      TOTSALE
4343.04 655.88 1814.25
```

C. 1.

```
      ∇READ DATA
[1]   ⎕FREAD 277,DATA
[2]   ∇
      ITEMS←2
      PRICES←3
      READ PRICES
11.52 3.72 6.15
```

Remember to use the comma between 277 and *DATA* in your function.

2.

```
      ∇TOTAL DATA
[1]   MAT←⎕FREAD 277,DATA
[2]   +/MAT
[3]   ∇
      TOTAL ITEMS
377 179 295
```

3.

```
          ∇TOTSALE ITEM
    [1]   IT←⎕FREAD 277 2
    [2]   PR←⎕FREAD 277 3
    [3]   IT[ITEM;]×PR[ITEM]
    [4]   ∇
          NUTS←1
          BOLTS←2
          WRENCHES←3
          TOTSALE NUTS
1267.2 1728 1347.84
          TOTSALE BOLTS
130.2 241.8 293.88
```

Notice that the function does not assign the data it read on lines [1] and [2] of the function to the variables *ITEMS* or *PRICES*. Remember, you assigned the number 2 to *ITEMS* and the number 3 to *PRICES* in Exercise C.1. You might want to use those variables again, so you do not want to confuse things by reassigning them now. To be really safe, you may want to assign the key variables inside the function.

```
          ∇ TOTSALE ITEM
    [1]   NUTS←1
    [2]   BOLTS←2
    [3]   WRENCHES←3
    [4]   IT←⎕FREAD 277 2
    [5]   PR←⎕FREAD 277 3
    [6]   IT[ITEM;]×PR[ITEM]
    [7]   ∇
```

This will ensure that the system will find what it needs when it looks for a value in *NUTS* or *BOLTS*. It takes slightly more computer time to run the function, however.

D. 1.

```
          'SALES' ⎕FCREATE 9
```

You can use any whole number that you are not already using.

2.

```
          SNOS ⎕FAPPEND 9
    1
          MON ⎕FAPPEND 9
    2
          SALES ⎕FAPPEND 9
    3
```

```
3.           ⎕FUNTIE 9
```

```
4.           ∇GETDATA
      [1]    'SALES' ⎕FTIE 1076
      [2]    SNOS←⎕FREAD 1076 1
      [3]    MON←⎕FREAD 1076 2
      [4]    SALES←⎕FREAD 1076 3
      [5]    ∇
```

5. Enter the function as shown.

```
6.           ∇MONTH NUM
      [1]    M←NUM
      [2]    ∇
```

```
7.           ∇PRINT
      [1]    (MON=M)/(SNOS=S)⌿SALES
      [2]    ∇
```

```
E.           ∇ GETNAME
      [1]    NAME←⎕FREAD 1068 2
      [2]    'NAME OF ITEM:'
      [3]    NEWNAME←20↑⍞
      [4]    NAME←NAME,[1]NEWNAME
      [5]    NAME ⎕FREPLACE 1068 2
      [6]    ∇

             ∇ GETDETAILS
      [1]    DETAILS←⎕FREAD 1068 3
      [2]    'SERIAL NUMBER, DATE PURCHASED, PRICE:'
      [3]    NEWD←35↑⍞
      [4]    DETAILS←DETAILS,[1]NEWD
      [5]    DETAILS ⎕FREPLACE 1068 3
      [6]    ∇

             ∇ INVSTART
      [1]    'INVENTOR' ⎕FTIE 1068
      [2]    ∇
```

F.

```
        ∇ BUDGET
[1]     'ENTER ITEMS (¨END¨ TO STOP):'
[2]     INAMES←20↑⍞
[3]     INAMES←1 20⍴INAMES
[4]     LP: 'NEXT ITEM:'
[5]     NEWLINE←20↑⍞
[6]     →(∧/'END'=3↑NEWLINE)⍴NUMS
[7]     NEWLINE←1 20⍴NEWLINE
[8]     INAMES←INAMES,[1]NEWLINE
[9]     →LP
[10]    NUMS:'ENTER BUDGET FIGURES FOR ',
(⍕1↑⍴INAMES),' ITEMS:'
[11]    N←⎕
[12]    TOT←+/N
[13]    PERCENT←100×N÷TOT
[14]    ' '
[15]    'BUDGET IS:'
[16]    ' '
[17]    F←'20A1,P<$>CF12.2,N<%>Q<%>F10.1'
[18]    F ⎕FMT (INAMES;N;PERCENT)
[19]    44 ⍴ '-'
[20]    T←20↑'TOTAL'
[21]    T←1 20⍴T
[22]    F ⎕FMT (T;+/N;100)
[23]    'DON''T FORGET TO )SAVE IF THIS IS THE
FINAL VERSION.'
[24]    ∇
```

Notice the variables are not localized because *SHOWBUDGET* needs the variables to run.

B

Error Messages

In using this book and typing in APL expressions, you may have seen error messages. Errors are easy to make — all you have to do is misspell a variable or function name or forget how to use a function. With APL, you can always correct your mistake and try again.

Table B-1 lists some of the common error messages, the probable cause, and a solution; your APL system may have error messages other than those listed here. Refer to your system documentation for a complete list.

Table B-1. Common APL Errors and Solutions

Error	Probable Cause	Solution
DEFN ERROR	A definition error occured when you are creating a function.	
	You tried to create a function with the same name as an existing function or variable.	Use *)FNS* or *)VARS* to see if the function exists. Erase the old function or use another name.
	You made an error with the del editor commands.	Check your typing of the commands.
	You tried to display a function that does not exist.	Check your spelling, or use *)FNS* to ensure that the function exists.

Table B-1. Continued

Error	Probable Cause	Solution
DOMAIN ERROR	You tried to use a function with inappropriate data; for example, using character data with a function defined for numeric data.	Use appropriate data with the function. You may need to check your documentation to see how to use the function properly.
	The answer is not defined for the function; for example, you cannot divide a number by zero.	
FILE ARGUMENT ERROR	You used too many letters in the file name.	Shorten the file name.
	You used illegal characters in the file name, or you used a number as the first character.	Change the file name.
FILE FULL	More additions to the file will exceed the storage reserved for it.	Use a different file, eliminate the file, drop unnecessary components in the file, or increase the storage reserved for the file.
FILE INDEX ERROR	You tried to read beyond the end of the file or before the beginning of the file.	Use *⎕FSIZE* to see how many components the file has.
FILE NAME ERROR	You tried to create a file with the same name as another file in the same library or directory.	Use *⎕FLIB* to list your files. Erase the old file or use a different name.
FILE NOT FOUND	The file does not exist in the specified library or directory.	Check the spelling of the file name. Use *⎕FLIB* to list the files in other libraries or directories.

Table B-1. Continued

Error	Probable Cause	Solution
FILE TIE ERROR	You are trying to use a tie number that has no file tied to it.	Tie the file and try again.
	You are trying to tie a file to a number already in use as a tie number.	Use ⎕*FNUMS* to see which numbers are in use. Tie the file with a different number.
FILE TIE QUOTA EXCEEDED	You have tied the maximum number of files allowed.	Use ⎕*FUNTIE* to untie unneeded files.
INCORRECT COMMAND	You used a system command incorrectly or without the correct arguments.	Check your typing. Check the command to be sure you included the necessary information.
INDEX ERROR	You used an invalid index (subscript) with a variable. For example, you may have specified the 11th element of a ten-element vector or the third column of a two-column matrix.	Check the shape of the variable and try again.
LENGTH ERROR	You are trying an operation with two variables whose shapes do not match. Or, you are trying to use a function with an argument of the wrong length.	Adjust the shapes of the variables with the reshape (ρ), take (↑), drop (↓), or ravel (,) functions. Or, use a variable with the appropriate length for the function.
LIMIT ERROR	You entered a number that is too large or too small for the system to handle. Or, the rank of your variable is too large for the system.	Represent the number or variable in a different way; for example, scale it or divide it into several parts.

Table B-1. Continued

Error	Probable Cause	Solution
NONCE ERROR	You used a feature that is not implemented on your APL system. (This condition is know as an error for the nonce.)	Use a different feature.
RANK ERROR	You used a function with data of incorrect rank; that is, the wrong number of dimensions. You may have tried to index a matrix without using a semicolon inside the brackets.	Check the shape of the data. Make sure the function you want to use accepts data with that shape. If you are indexing a matrix, be sure to include a semicolon between the row and column indices inside the brackets.
STACK FULL	The function you are running is caught in an endless loop and is calling another function repeatedly (recursively).	Check your function for problems with loops or reduce the number of recursive calls.
SYMBOL TABLE FULL	You used too many names for your functions and variables. (These names are called symbols.) The system has no space for more names.	Use *)SYMB* to determine the allowable number of symbols in the symbol table. You must reset the symbol table in a clear workspace. Enter the following commands to set the maximum number of symbols to 1024:)SIC)SAVE TEMP)CLEAR)SYMBOLS 1024)COPY TEMP)SAVE MYWS)DROP TEMP

Table B-1. Continued

Error	Probable Cause	Solution
SYNTAX ERROR	The APL expression does not conform to APL syntax rules.	Check your typing first.
	You used an uneven numbers of brackets or parentheses.	Insert a matching bracket or parenthesis.
	You used the wrong number of arguments for the function.	Check the function to see whether it is monadic (one argument), dyadic (two arguments), or niladic (no arguments).
	You did not put a function between two arguments.	Insert a function name between the two arguments.
VALUE ERROR	You typed the name of a function or variable that does not exist in the active workspace.	Use *)FNS* or *)VARS* to see a list of the functions and variables in the workspace. Be sure you typed the name correctly.
	You tried to use the result of a user-defined function that was not defined with an explicit result.	Change the definition of the function so that it returns an explicit result.
WS FULL	You tried an operation that required more space than is available in the active workspace.	Try erasing some unneeded variables or user-defined functions to free some space. Localize variables within user-defined functions. Place variables and functions in files.

Table B-1. Continued

Error	Probable Cause	Solution
WS NOT FOUND	The workspace does not exist in the specified library or directory.	Check your spelling. Use ⎕*WSLIB* to list the workspaces in your libraries and directories.

C

APL Programs Used in This Book

```
     ∇ ACCOUNT V
[1]    ⌈/V
[2]    ⌊/V
[3]    +\V
[4]    +/V
     ∇
```

```
     ∇ ACCOUNT2 DATA
[1]    'THE LARGEST DEPOSIT IS ',⍕⌈/DATA
[2]    'THE LARGEST WITHDRAWAL IS ',⍕⌊/DATA
[3]    'THE RUNNING BALANCE IS ',⍕+\DATA
     ∇
```

```
     ∇ R←A ADD B
[1]    R←A+B
     ∇
```

```
     ∇ ADDER;NOS
[1]    'ENTER NUMBERS TO BE ADDED'
[2]    NOS←⎕
[3]    'THE TOTAL IS ',(⍕+/NOS),'.'
     ∇
```

```
      ∇ ADDREC
[1]     'ENTER YOUR RECIPE LINE BY LINE (USE "END" TO
        STOP):'
[2]     RECIPE←60↑⍞
[3]     RECIPE←1 60⍴RECIPE
[4]    L1:'NEXT LINE:'
[5]     NEWLINE←60↑⍞
[6]     →(∧/'END'=3↑NEWLINE)⍴L2
[7]     NEWLINE←1 60⍴NEWLINE
[8]     RECIPE←RECIPE,[1]NEWLINE
[9]     →L1
[10]   L2:RECIPE ⎕FAPPEND 1111
[11]    DIR←⎕FREAD 2222 1
[12]    NEWENTRY←RECIPE[1;]
[13]    DIR←DIR,[1]NEWENTRY
[14]    DIR ⎕FREPLACE 2222 1
[15]    'NEW RECIPE FILED.'
      ∇
```

```
      ∇ ASKAVERAGE
[1]     'WHAT NUMBER DO YOU WANT TO AVERAGE?'
[2]     VECTOR←⎕
[3]     ANSWER←(+/VECTOR)÷⍴VECTOR
[4]     'THE ANSWER IS ',(⍕ANSWER),'.'
      ∇
```

```
      ∇ ASKDICE
[1]     +/? 6 6
[2]     'TRY AGAIN? (YES OR NO):'
[3]     →('Y'=1↑⍞)/1
      ∇
```

```
      ∇ ASKDICE2
[1]    LINE1:+/? 6 6
[2]     'TRY AGAIN? (YES OR NO):'
[3]     →('Y'=1↑⍞)⍴LINE1
      ∇
```

```
    ∇ ASKDIE
[1]  L1:?6
[2]   'TRY AGAIN? (YES OR NO):'
[3]   →('Y'=1↑⍞)⍴L1
    ∇
```

```
    ∇ ASKDIE2
[1]  LINE1:?6
[2]   'TRY AGAIN? (YES OR NO):'
[3]   →('Y'=1↑⍞)⍴LINE1
    ∇
```

```
    ∇ ASKHIST;NUMS
[1]   'ENTER FOUR NUMBER FOR BAR CHART'
[2]   NUMS←⎕
[3]   HIST NUMS
    ∇
```

```
    ∇ ASKREPEAT;THING;X
[1]   'ENTER THING TO BE REPEATED:'
[2]   THING←⍞
[3]   'REPEAT HOW MANY TIMES?'
[4]   X←⎕
[5]   X⍴THING
    ∇
```

```
    ∇ AVG DATA
[1]   (+/DATA)÷⍴DATA
    ∇
```

```
    ∇ R←AVG2 A
[1]   R←(+/A)÷⍴A
    ∇
    ∇ BUDGET
[1]     'ENTER BUDGET ITEM (¨END¨ TO STOP):'
[2]     INAMES←20↑⍞
[3]     INAMES←1 20⍴INAMES
[4]    LP:'NEXT ITEM:'
[5]     NEWLINE←20↑⍞
[6]     →(∧/'END'=3↑NEWLINE)⍴NUMS
[7]     NEWLINE←1 20⍴NEWLINE
[8]     INAMES←INAMES,[1]NEWLINE
[9]     →LP
[10]   NUMS:'ENTER FIGURES FOR ',(⍕1↑⍴INAMES),' ITEMS'
[11]    N←⎕
[12]    TOT←+/N
[13]    PERCENT←100×N÷TOT
[14]    ''
[15]    'BUDGET IS:'
[16]    ''
[17]    FSB←'20A1,P<$>CF10.2,N<%>Q<%>F10.1'
[18]    FSB ⎕FMT (INAMES;N;PERCENT)
[19]    44⍴'¯'
[20]    T←20↑'TOTAL'
[21]    T←1 20⍴T
[22]    FSB ⎕FMT (T;+/N;100)
[23]    'DON''T FORGET TO )SAVE IF THIS IS THE FINAL
        VERSION.'
    ∇
```

```
    ∇ BUILDDIRECTORY
[1]     DIR←⎕FREAD 2222 1
[2]     'ENTER TITLE OF RECIPE:'
[3]     NEWENTRY←60↑⍞
[4]     NEWENTRY←1 60⍴NEWENTRY
[5]     DIR←DIR,[1]NEWENTRY
[6]     DIR ⎕FREPLACE 2222 1
    ∇
```

```
    ∇ CALC
[1]     INCOMEDATA[3;]←INCOMEDATA[1;]-INCOMEDATA[2;]
    ∇
```

```
    ∇ CHECK BUCKS
[1]     WITHD←WITHD,BUCKS
    ∇
```

```
     ∇ A CHOOSE B
[1]   A[B]
     ∇


     ∇ COLTOT MAT
[1]    +⌿MAT
     ∇

     ∇ COUNT
[1]    ONE
[2]    TWO
[3]    TOO
[4]    TREE
[5]    FORE
     ∇


     ∇ DEPOSIT BUCKS
[1]    DEPOS←DEPOS,BUCKS
     ∇


     ∇ DIE
[1]    ?6
     ∇


     ∇ ANS←DIE2
[1]    ANS←?6
     ∇


     ∇ ENTER
[1]    GETNAME
[2]    GETDETAILS
[3]    'NEW ITEM FILED.'
     ∇


     ∇ R←A EQ B
[1]    R←(A=B)/B
     ∇
     ∇ ERASEBLANK S
[1]     (' '≠S)/S
     ∇


     ∇ FILERECIPE
[1]     'RECIPES' ⎕FTIE 9971
[2]     'ENTER YOUR RECIPE LINE BY LINE (USE "END" TO
        STOP):'
[3]     RECIPE←60↑⍞
[4]     RECIPE←1 60⍴RECIPE
[5]   L1:'NEXT LINE:'
[6]     NEWLINE←60↑⍞
[7]     →(∧/'END'=3↑NEWLINE)⍴L2
[8]     NEWLINE←1 60⍴NEWLINE
[9]     RECIPE←RECIPE,[1]NEWLINE
[10]    →L1
[11]  L2:RECIPE ⎕FAPPEND 1111
[12]    ⎕FUNTIE 9971
[13]    'DONE. NEW RECIPE FILED.'
     ∇


     ∇ FILESALES;SALES
[1]     ⍝ PUT NEW SALES FIGURES IN SALES FILE
[2]     'SALESFIG' ⎕FTIE 2201
[3]     'ENTER SALES FIGURES FOR THIS WEEK'
[4]     SALES←⎕
[5]     SALES ⎕FAPPEND 2201
[6]     ⎕FUNTIE 2201
[7]     'NEW SALES FILED.'
     ∇


     ∇ FLIP
[1]     N←2 5⍴'HEADSTAILS'
[2]     ((?2)=1 2)⌿N
[3]     'TRY AGAIN? (YES OR NO):'
[4]     →('Y'=1↑⍞)⍴2
     ∇
```

```
    ∇ FLIP2
[1]   N←2 5ρ'HEADSTAILS'
[2]  MORE:((?2)=1 2)⌿N
[3]   'TRY AGAIN? (YES OR NO):'
[4]   →('Y'=1↑⍞)ρMORE
    ∇
```

```
    ∇ GETDATA
[1]   'SALES' ⎕FTIE 1076
[2]   SNOS←⎕FREAD 1076 1
[3]   MON←⎕FREAD 1076 2
[4]   SALES←⎕FREAD 1076 3
    ∇
```

```
    ∇ GETDETAILS
[1]   DETAILS←⎕FREAD 1068 3
[2]   'SERIAL NUMBER, DATE PURCHASED, AND PRICE:'
[3]   NEWD←35↑⍞
[4]   DETAILS←DETAILS,[1]NEWD
[5]   DETAILS ⎕FREPLACE 1068 3
    ∇
```

```
    ∇ GETNAME
[1]   NAME←⎕FREAD 1068 2
[2]   'NAME OF ITEM:'
[3]   NEWNAME←20↑⍞
[4]   NAME←NAME,[1]NEWNAME
[5]   NAME ⎕FREPLACE 1068 2
    ∇
```

```
    ∇ R←A GT B
[1]   R←(A>B)/A
    ∇
```

```
      ∇ GUESS
[1]     'I''VE PICKED A NUMBER BETWEEN 1 AND 3.'
[2]     'WHAT IS IT?'
[3]    LOOP:'ENTER YOUR NUMBER:'
[4]     NUM←⎕
[5]     ANS←?3
[6]     →(ANS=NUM)ρOK
[7]     'I''M SORRY. THE NUMBER WAS ',(⍕ANS),'.'
[8]     'TRY AGAIN?'
[9]     →('Y'=1↑⍞)ρLOOP
[10]    →0
[11]   OK:'RIGHT! TRY AGAIN?'
[12]    →('Y'=1↑⍞)ρLOOP
      ∇

      ∇ R←HALF NUM
[1]     R←NUM×.05
      ∇


      ∇ HELLO
[1]     'MY NAME IS JERRY.  WHAT''S YOURS?'
[2]     A←⍞
[3]     'NICE TO MEET YOU, ',A,'.'
      ∇


      ∇ HIST D
[1]     D[1] OF '⎕'
[2]     D[2] OF '⎕'
[3]     D[3] OF '⎕'
[4]     D[4] OF '⎕'
      ∇


      ∇ INIBAL AMOUNT
[1]     DEPOS←AMOUNT
[2]     WITHD←0
      ∇


      ∇ INVSTART
[1]     'INVENTOR' ⎕FTIE 1068
      ∇
```

```
    ∇ LIST;DIR;NUMS;NUMBER;NAMES;NAM;DET
[1]   ⍝ MANAGES HOUSEHOLD INVENTORY SYSTEM
[2]   L1:'LIST, ITEMS, OR QUIT? (D,I,Q):'
[3]    →('DIQ'=1↑⍞)/DI,ITEMS,EXIT
[4]    'PLEASE ENTER D, I, OR Q'
[5]    →L1
[6]   DI:DIR←⎕FREAD 1068 2
[7]    NUMS←⍳1↑⍴DIR
[8]    'Q<  >I5,20A1' ⎕FMT (NUMS;DIR)
[9]    →L1
[10]  ITEMS:'LIST WHICH ITEM (BY NUMBER)?'
[11]   NUMBER←⎕
[12]   NAMES←⎕FREAD 1068 2
[13]   DETAILS←⎕FREAD 1068 3
[14]   NAM←NAMES[NUMBER;]
[15]   DET←DETAILS[NUMBER;]
[16]   NAM,DET
[17]   →L1
[18]  EXIT:
    ∇


    ∇ LISTDIRECTORY
[1]    DIR
    ∇


    ∇ R←A LT B
[1]    R←(A<B)/A
    ∇


    ∇ MONTH NUM
[1]    M←NUM
    ∇


    ∇ NEWBAL
[1]    (+/DEPOS)-+/WITHD
    ∇
```

```
     ∇ NOBLANK CHARVECT
[1]   ⍝ REMOVE BLANKS FROM CHARACTER VECTOR
[2]    (CHARVECT≠' ')/CHARVECT
     ∇
```

```
     ∇ X OF Y
[1]    XρY
     ∇
```

```
     ∇ R←WIDTH PERIM LENGTH
[1]    R←2×WIDTH+LENGTH
     ∇
```

```
     ∇ WIDTH PERIMETER LENGTH
[1]    2×WIDTH×LENGTH
     ∇
```

```
     ∇ PRINT
[1]    (MON=M)/(SNOS=S)⌿SALES
     ∇
```

```
     ∇ READ DATA
[1]    ⎕FREAD 277,DATA
     ∇
```

```
     ∇ READ2
[1]    'RECIPES' ⎕FTIE 3371
[2]    'WHAT IS THE NUMBER OF THE RECIPE?'
[3]    ANS←⎕
[4]    ⎕FREAD 3371,ANS
[5]    ⎕FUNTIE 3371
     ∇
```

```
     ∇ READREC
[1]    'READ WHICH RECIPE (BY NUMBER)'
[2]    RECNO←⎕
[3]    ⎕FREAD 1111,RECNO
     ∇
     ∇ RECIPES
[1]    TIEFILES
[2]   L0:'LIST, READ, ADD, OR QUIT? (L, R, A, OR Q):'
[3]    →('LRAQ'=1↑⍞)/L1,L2,L3,L4
[4]   L1:LISTDIRECTORY
[5]    →L0
[6]   L2:READREC
[7]    →L0
[8]   L3:ADDREC
[9]    →L0
[10]  L4:⎕FUNTIE 1111 2222
     ∇
```

```
     ∇ REPEAT X
[1]    X,' ',X,' ',X,' ',X,' ',X
     ∇
```

```
     ∇ REPORT
[1]    '               HOMEGROWN CO.'
[2]    ''
[3]              1989          1990          1991
[4]    ''
[5]    CALC
[6]    '7A1,3CF12.2' ⎕FMT (NAMES;INCOMEDATA)
     ∇
```

```
     ∇ REPT DATA
[1]    TOT←+/DATA
[2]    '9A1,3CP<$>F12.2' ⎕FMT (N1;DATA;TOT)
     ∇
```

```
     ∇ ROWTOT MAT
[1]    +/MAT
     ∇
```

```
     ∇ RPT DATA
[1]    DATA
[2]    '______________'
[3]    COLTOT DATA
     ∇
```

```
      ∇ RPT2 DATA
[1]     '                     SALES FIGURES'
[2]     '               WEEK1            WEEK2            TOTALS'
[3]     TOT←+/DATA
[4]     '9A1,3CP<$>F12.2' ⎕FMT (N1;DATA;TOT)
      ∇


      ∇ RUNTOT DATA
[1]     +\DATA
      ∇

      ∇ SALESMAN NUM
[1]     S←NUM
      ∇


      ∇ SHOWBUDGET
[1]   ⍝ DISPLAYS BUDGET CREATED BY ¨BUDGET¨ FUNCTION
[2]     ''
[3]     FSB ⎕FMT (INAMES;N;PERCENT)
[4]     44⍴'¯'
[5]     FSB ⎕FMT (T;+/N;100)
      ∇


      ∇ SQUARE X
[1]     'THE NUMBER SQUARED IS ',⍕X×X
      ∇


      ∇ TIE
[1]     'EXERCISE ⎕FTIE 277
      ∇


      ∇ TIEFILES
[1]     'RECIPES' ⎕FTIE 1111
[2]     'DIRECTRY' ⎕FTIE 2222
      ∇
```

```
     ∇ TOTAL COMPNO
[1]    MAT←⎕FREAD 277,COMPNO
[2]    +/MAT
     ∇
```

```
     ∇ TOTSALES ITEM
[1]    IT←⎕FREAD 277 2
[2]    PR←⎕FREAD 277 3
[3]    IT[ITEM;]×PR[ITEM]
     ∇
```

```
     ∇ TOTSALES
[1]    ITEMS←⎕FREAD 277 2
[2]    PRICES←⎕FREAD 277 3
[3]    PRICES×+/ITEMS
     ∇
```

Glossary

- **Append**
  Add data to the end of a file. You normally add new data to a file by appending the data to the file (that is, by using the function ⎕*FAPPEND*).

- **Character Data**
  Data that is not used for arithmetic. Character data is often used in APL functions to prompt the user for an entry or to label data. You distinguish character data from numeric data by placing single quotes at the beginning and end of the data.

- **Character Input Mode**
  State in which the computer accepts character data *without* having the data enclosed in single quotes. Used in interactive functions. Also called quote-quad input.

- **Column**
  The second dimension of a matrix. The vertical arrangement of the numbers in a matrix.

- **Component**
  A single segment of a file. Components are numbered sequentially (1, 2, 3, and so on) and can contain any type of data array that your APL system supports.

- **Data**
  Numbers, letters, and symbols that a function uses to do work. APL works with data in the form of scalars, vectors, matrices, and multi-dimensional arrays.

- **Del**
  The ∇ symbol, which identifies the beginning and end of a function definition.

- **Dyadic Function**
  Type of function that has two arguments, or pieces of data. In particular, it has both a left and a right argument. An example is the plus function: 4+5. See also Monadic Function.

- **Element**
  An item in a vector, matrix, or higher-rank array. For example, the third elements of the vectors 1 2 3 4 and *'MARY'* are 3 and *'R'*.

- **Evaluated Input Mode**
  State in which the system accepts input in the form of numbers, characters enclosed in single quotes, or APL expressions that produce results. It is used in interactive functions and is also called quad input mode.

- **Explicit Result**
  A function result that the system stores so that it can be used by other functions for further work. Functions *without* explicit results display, but do not store, their results.

- **Expression**
  See Statement.

- **File**
  Storage place for data that is independent of the workspace. Files must be tied to be active, and they are referred to by their tie numbers.

- **Formatting**
  Process of displaying system responses (output) in a form different from the one the system usually provides. This includes changing the spacing, inserting commas and other symbols or text, and changing the number of decimal places.

- **Function**
  List of instructions for performing a task. In APL, functions can be built-in (such as + and ÷) or user-defined. Both types of functions use data to do work.

- **Function Definition Mode**
  State in which the system accepts the lines of a user-defined function. The system collects these lines and stores them until you run the function. You enter and leave this state by entering a del symbol (∇).

- **Global Variables**
  Variables that are used throughout the workspace. These variables might be used by a number of functions in the workspace. See also Local Variables.

- **Immediate Execution Mode**
  State in which the computer acts as a desk calculator, processing APL statements and giving answers. When you start APL, the system is in this mode.

- **Interactive**
  Type of function that interacts with you by asking you questions and working with the data you enter.

- **Iota**
  The ⍳ symbol, used by the count function.

- **Length**
  The number of elements in a vector. For example, both of the vectors 1 2 3 4 and *'MARY'* have a length of four.

- **Line Labels**
  Names used to identify lines of a user-defined function. Use line labels together with branching to guide the system's execution of a function. Use a colon to separate labels from APL statements on the same line of the function.

- **Local Variables**
  Variables that are locked inside a user-defined function. These variables exist only while a function is being run; the system erases them when the function completes. See also Global Variables.

- **Matrix**
  A table of data. A matrix has two dimensions — rows and columns. Rows are the first dimension, and columns are the second dimension. You can use the reshape function (ρ) to change both numeric data and character data into matrices.

- **Monadic Function**
  Type of function with a right argument only. That is, a function that takes data on the right side only. An example is the shape function: `ρVECTOR`. See also Dyadic Function.

- **Numeric Data**
  Data the system can use for arithmetic (numbers). See also Character Data.

- **Order of Execution**
  The rule APL uses to process an expression. APL always begins with the part of the expression farthest to the right, solves it, and uses the answer to solve the next part of the expression. The system stops when it reaches the left-most part of the expression. APL then either displays the final result or stores the result in a variable, depending on what you have told it to do.

- **Quad**
  The ⎕ symbol. Used in function editing commands, in file system functions and other system functions, and to request evaluated input.

- **Quad Input Mode**
  See Evaluated Input Mode.

- **Quote-quad**
  The ⍞ symbol, used to request character input.

- **Quote-quad Input Mode**
  See Character Input Mode.

- **Relational Functions**
  Built-in APL functions that show a relationship between two numbers. Includes the functions less than (<), less than or equal to (≤), equals (=), greater than or equal to (≥), greater than (>) and not equal to (≠).

- **Repetition Factor**
  A number you place at the beginning of a format phrase to indicate how many times the system should repeat the format phrase. That is, it shows how many times the system should use the same format phrase to format consecutive columns of data.

- **Rho**
  The ρ symbol. Used for the functions shape and reshape.

- **Row**
  The horizontal arrangement of numbers in a matrix. Rows are the first dimension of a matrix.

- **Scalar**
  A single number. See also Vector and Matrix.

- **Statement**
  String of APL symbols and numbers or letters. A statement always contains at least one function (a symbol) and some data (numbers or letters).

- **Subfunction**
  Function called by another function. The main function includes the name of the subfunction as one of its lines; thus the system runs the subfunction as part of running the main function.

- **Syntax**
  General form of a function. A function's syntax tells how many arguments a function takes and which side of the function name or symbol they appear on. Functions have three possible syntaxes — one argument (a right argument), two arguments (left and right arguments), or no arguments.

- **System Commands**
  Commands such as *)LOAD* and *)SAVE* that control the system rather than operate on arrays.

- **System Functions**
  Commands such as *⎕LOAD* that are similar to system commands but can be used inside user-defined functions. Some system functions, such as *⎕FMT*, have capabilities beyond those of system commands.

- **Tie Number**
  Whole number associated with a file while the file is active. You must use unique tie numbers for each file you have tied, but you can use different tie numbers for a file each time you tie it.

- **Variable**
  Storage place for data. Variables have names that can include both letters and numbers, as long as the first character of the name is a letter. Data stored in variables is stored *permanently* when you enter the command *)SAVE*.

- **Vector**
  List of numbers, each separated by at least one blank space. Or, list of characters enclosed in quotes. For example, 7 9 11 is a vector.

- **Workspace**
  Place for storing both functions and data. Workspace names must begin with a letter. The system permanently stores workspaces when you enter the command *)SAVE*.

Index

READER COMMENT CARD

We welcome your evaluation of this manual. Your comments and suggestions help us improve our publications.

Title of the manual you are evaluating: APL Is Easy!

Please circle one number for each.	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
• The manual is well organized.	1	2	3	4	5
• The manual is easy to understand.	1	2	3	4	5
• The manual is complete.	1	2	3	4	5
• The manual is clearly written.	1	2	3	4	5
• I can find the information I want.	1	2	3	4	5
• Concepts and vocabulary are easy to understand.	1	2	3	4	5
• Examples are clear and useful.	1	2	3	4	5
• The manual contains enough examples.	1	2	3	4	5
• Illustrations are clear and helpful.	1	2	3	4	5
• The manual contains enough illustrations.	1	2	3	4	5
• The index is thorough.	1	2	3	4	5
• Layout and format enhance the manual's usefulness.	1	2	3	4	5
• This manual meets my overall expectations.	1	2	3	4	5

(over, please)

Please write additional comments, particularly if you disagree with a statement above. Use additional pages if you wish. The more specific your comments, the more useful they are to us.

Comments:

Although optional, we would appreciate the following information.

Name

Title

Company

Address

City/State/Zip

Country

Phone

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT #10586 ROCKVILLE, MD

Postage Will Be Paid By Addressee

STSC, Inc.
ATTN: Manager, APL★PLUS Documentation
2115 East Jefferson Street
Rockville, MD 20852 U.S.A.